UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

PROGRAMACIÓ CONSCIENT DE L'ARQUITECTURA DEL MAQUINARI:

de l'anàlisi acadèmica a l'aplicació professional

Autor Berenguer Montserrat Robert

Director Òscar Amorós Huguet

Realitzat a Departament de matemàtica aplicada i anàlisi

Barcelona, 27 de juny de 2018

Abstract

Architecture-aware programming is a type of programming where the principal goal is the algorithm optimization (reducing the executing time) considering the used architecture and what the compiler can do on its own.

This project deals with this type of programming from three very different points of view, all of them extremely related. It begins with the design and build of a simple and programmable architecture in order to deepen the knowledge of design and build of a processors' architecture. Continues with the creation of an academical example where different architecture-aware programming techniques are applied in order to optimize a primer number finder algorithm; it uses two different architectures so as to demonstrate which one is more appropriate. Finally, it applies the acquired knowledge in this scope to a professional environment to migrate an edge computing application to a more low cost embedded platform.

Resum

La programació conscient de l'arquitectura és un tipus de programació on l'objectiu principal és la optimització (reducció del temps d'execució) d'algorismes tenint en compte l'arquitectura que s'utilitza i el que el compilador pot fer.

Aquest projecte tracta aquest tipus de programació des de tres punts de vista molt diferents, però tots ells relacionats. Comença amb el disseny i muntatge d'una arquitectura programable molt bàsica per tal d'aprofundir en el coneixement del disseny i construcció d'arquitectures de processadors. Segueix amb la creació d'un exemple acadèmic, on s'apliquen diferents tècniques de la programació conscient de l'arquitectura per optimitzar un algorisme de cerca de nombres primers; s'utilitzen dues arquitectures molt diferents per demostrar quina és la més adequada. I finalment, aplica els coneixements adquirits en aquest àmbit a l'entorn professional per tal de migrar una aplicació d'*edge computing* a una plataforma encastada de baix cost.

Resumen

La programación consciente de la arquitectura es un tipo de programación donde el objetivo principal es la optimización (reducción del tiempo de ejecución) de algoritmos teniendo en cuenta la arquitectura que se utiliza y lo que el compilador puede hacer.

Este proyecto trata este tipo de programación desde tres puntos de vista muy distintos, pero todos ellos relacionados. Empieza con el diseño y montaje de una arquitectura programable muy simple para profundizar en el conocimiento del diseño y construcción de procesadores. Sigue con la creación de un ejemplo académico, dónde se aplican diferentes técnicas de la programación consciente de la arquitectura para optimizar un algoritmo de búsqueda de números primos; se utilizan dos arquitecturas muy distintas para demostrar cual de ellas es la más adecuada. Y finalmente, aplica los conocimientos adquiridos en este ámbito al entorno profesional para migrar una aplicación de *edge computing* a una plataforma embebida de bajo coste.

Índex

Capítol 1

Introducció i motivació

La programació de computadors consisteix en escriure ordres per tal de construir un programa que satisfaci certes necessitats. Aquestes ordres seran interpretades i executades per un maquinari. Però el maquinari *parla una llengua diferent a la humana*, i és per això que al llarg del temps, i també actualment, s'han desenvolupat eines per tal d'abstraure tots els detalls més escabrosos d'aquest llenguatge màquina i fer-lo més accessible al programador.

Aquests programes que abstrauen tots aquests detalls al programador són els compiladors, i, en certa mesura, els sistemes operatius. Tot i que s'ha avançat moltíssim en el seu disseny i capacitats, no són prou intel·ligents perquè sent inconscient de l'arquitectura del maquinari, puguin generar codi tant eficient com si se'n tingués certa coneixença. Només tenint cert discerniment del funcionament del compilador i les seves nombroses opcions, de com està organitzat el maquinari sobre el qual s'executarà el programa i en quin sistema operatiu s'executarà, es poden escriure codis molt més eficients.

A més a més, sempre m'ha cridat força l'atenció la programació que trenca la barrera de la seqüencialitat: la programació concurrent. Aquest tipus de programació té per objectiu millorar el rendiment, però per si sol pot no ser suficient per assolir-ne el nivell desitjat. Per tant, per fer una bona programació concurrent és necessari tenir consciència del maquinari sobre el qual s'executarà el programa.

Podríem dir que en la gran majoria de casos, l'eficiència no és un aspecte important pel funcionament del programa, més enllà de les bones pràctiques habituals; podríem pensar en programaris de gestió de documents, aplicacions mòbils senzilles, aplicacions web, etc. Però hi ha casos en els que el rendiment que s'obté habitualment no és suficient, i cal treballar de forma dedicada aquest aspecte; podríem parlar d'aplicacions d'anàlisi visual en temps real (cotxes autònoms), gestors de gran quantitat de dades, xarxes neuronals, càlculs matemàtics i programari pensat per executar-se en supercomputadors, com els simuladors meteorològics o analitzadors moleculars. La consciència de l'arquitectura, també permet

fer-ne una bona selecció per utilitzar-les, en funció dels algorismes a implementar i de les característiques del problema a solucionar.

És justament en aquests últims casos on el programador requereix tenir certa consciència sobre el maquinari que executarà el codi. Com modificar el comportament del compilador, com optimitzar la localitat de dades, com forçar certes variables a ser emmagatzemades en registres són alguns dels principals aspectes del que tradicionalment s'ha dit *programació conscient de l'arquitectura*.

Vull aprofundir en aquest àmbit de programació, i ho faré des del punt de vista didàctic, on exposaré algorismes paral·lels en un entorn controlat i també des del punt de vista pràctic, on abordaré alguns problemes que té una aplicació de visió artificial en temps real que ja està en producció.

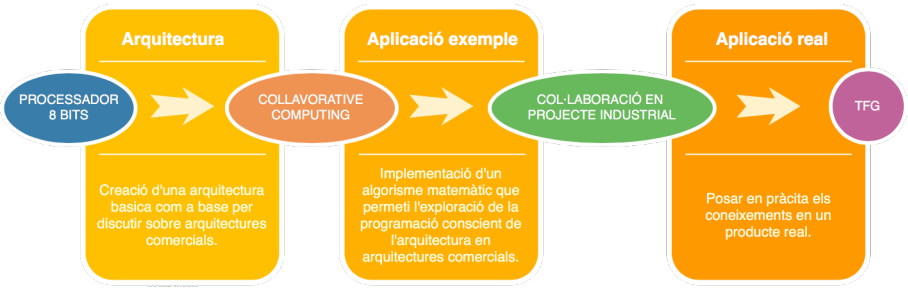

Figura 1.1: Diagrama de l'organització del treball realitzat pel TFG.

A la figura 1.1 es poden veure els passos que seguiré per realitzar el treball. Com es pot apreciar, consta de tres parts diferenciades, però relacionades entre sí. Per facilitar-ne la lectura, la memòria està separada també en tres parts; cadascuna consta de de la seva introducció i objectius, els seus resultats i les seves conclusions. Per aquest motiu, els capítols dedicats als objectius i conclusions seran més generals i el capítol de resultats, l'he omès.

El primer pas serà dissenyar i construir des de zero una nova arquitectura de maquinari, basada en l'arquitectura Von Neumann que sigui capaç de resoldre problemes aritmètics molt senzills. Fent-ho, em permetrà entendre les decisions de dissney i les particularitats de les diferents arquitectures comercials, especialment CPU i GPU, el perquè de les seves diferències i per a quin tipus de software és més adequada, per tal de fer una bona programació conscient de l'arquitectura.

El segon pas serà adaptar un programa de càlcul numèric que vaig fer fa un temps perquè pugui ser executat de forma col·laborativa entre la CPU i la GPU per obtenir els millors temps d'execució possibles.

I finalment, per tal d'entendre les estratègies i necessitats del món professional respecte la programació conscient de l'arquitectura, he estat treballant al departament de Recerca i Desenvolupament de Mediapro en un projecte on s'estan explorant de noves plataformes, així com de les diferències en el codi en funció del sistema operatiu i de la plataforma.

Capítol 2

Planificació

Aquest treball l'he plantejat en 3 seccions: el disseny de l'arquitectura, l'exploració acadèmica de la programació orientada a l'arquitectura i l'aplicació d'aquesta tècnica al món professional. Algunes d'aquestes tasques les vaig començar bastant abans de la matriculació del treball de final de grau. La figura 2.1 ho esquematitza.

Llegenda	Tasca
G1	**Disseny d'una arquitectura**
1	Disseny del processador de 8 bits
2	Muntatge del processador de 8 bits
G2	**Programació conscient de l'arquitectura aplicada**
3	Creació del projecte i re-escriptura dels algorismes
4	Exploració de noves arquitectures i optimització
G3	**Programació conscient de l'arquitectura a l'empresa**
5	Canvi de sistema operatiu: de Windows a Linux
6	Canvi de plataforma: de x86_64 a NVIDIA Jetson TX2 (ARM)
7	Exploració de la llibreria multimèdia GStreamer
8	Exploració de nous *frameworks* per *edge computing* a la Jetson
9	Optimització del *kernel* de correcció de distorció i color

Taula 2.1: Llegenda pel diagrama de la figura 2.1

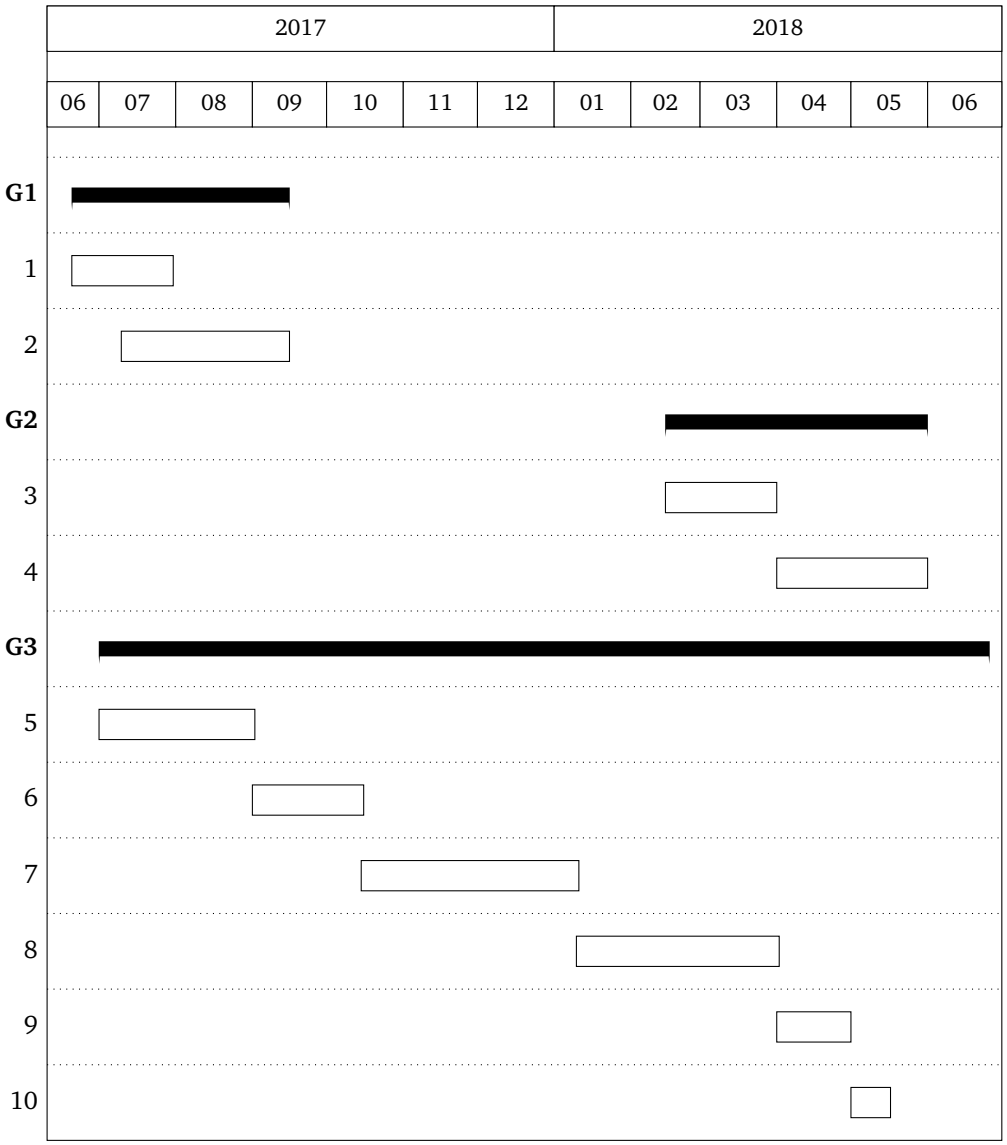

Figura 2.1: Diagrama de Gantt de l'organització de les tasques pel treball.

Capítol 3

Objectius

Aquest treball consta de tres parts diferencides, lligades a un objectiu transversal, que és aprofundir en el coneixement de la programació conscient de l'arquitectura. Cadascuna d'aquestes parts, però, té uns objectius particulars:

- La primera, té l'objectiu d'aprofundir en el coneixement del disseny i construcció d'arquitectures de processadors, així com analitzar les diferències arquitectòniques entre les CPU i les GPU.
- La segona, té el de crear un exemple acadèmic per aplicar el concepte de la programació conscient de l'arquitectura i entendre el motiu pel qual fa que un algorisme sigui més eficient executat en una arquitectura tipus CPU o tipus GPU.
- I finalment, la tercera, té l'objectiu d'entendre com s'aplica el concepte de la programació conscient de l'arquitectura en el món professional: saber què requereix, tant per part dels enginyers com per part de negoci, quin benefici aporta al producte i quina planificació a nivell d'equip és necessària per aconseguir-ho.

Aquests són els objectius de les tres parts del treball. Cada part, però, té objectius secundaris que estan exposats a la corresponent introducció.

Capítol 4

Desenvolupament

4.1 Disseny d'una arquitectura

En aquesta secció exploraré el disseny i creació d'una arquitectura de processador. Explicaré les diferents parts de què es composa, com estan construïdes a nivell electrònic i posaré un èmfasi especial a la creació de la ISA (*Instruction Set Architecture*) per aquest processador. Finalment, mostraré un petit programa fent servir un llenguatge assemblador propi que es pot executar en el prototip que he muntat.

4.1.1 Motivació

En aquest apartat dissenyaré una arquitectura de processador a partir del model d'arquitectura Von Neumann. Serà una arquitectura de 8 bits molt semblant a les dels primers processadors que es van construir. Crearé també un conjunt d'instruccions (ISA) seguint la filosofia RISC (*Reduced Instructsion Set Computer*). Així, aconseguiré una arquitectura senzilla, ja que a part de dissenyar-la, també la muntaré amb components electrònics.

Fent-ho així, podré tenir una idea més formada del procés i les decisions de disseny i construcció d'un processador i també explorar els principis més bàsics de la programació conscient de l'arquitectura.

Finalment, un dels objectius secundaris a complir és que aquest processador pugui introduir els estudiants de batxillerat tecnològic al món de l'Enginyeria Informàtica i l'Enginyeria Electrònica, així que un requeriment per complir-ho és que sigui didàctic i fàcil d'entendre.

4.1.2 Antecedents

Al subapartat anterior he estat parlant de dissenyar i muntar un processador de 8 bits. Existeixen nombrosos projectes adreçats principalment a l'aprenentatge del disseny de processadors. Alguns són llibres, d'altres estan en un format de blog o fins i tot, en format vídeo. Alguns dels que més s'acosten al que vull fer són:

- Zavala et al. [17] mostren com es dissenya un processador de 8 bits d'arquitectura RISC amb la intensió de formar nous alumnes universitaris en aquesta matèria, argumentant que el disseny de processadors és una habilitat indispensable actualment. Ensenyen alguns diagrames interessants sobre alguns components escencials, com un banc de registres o una ALU (*Arithmetic Logic Unit*). Plantegen també un conjunt d'instruccions, però sense entrar gaire en detall.
- El llibre de Malvino i Brown [8] és una de les referències principals sobre el disseny d'arquitectures. En proposa tres, anomenades SAP (*Simple As Possible*), en ordre creixent de complexitat, explicant la funció de cada component i com interactuen entre ells. Fa algunes referències a circuits integrats que encapsulen certes funcions i dóna algunes pistes pel posterior muntatge.
- El seguit d'instruccions online, de l'autor de sobrenom spel3o [14], segueix el model de Malvino i Brown per construir l'arquitectura bàsica (SAP-1) amb components electrònics conectats entre sí mitjançant plaques de prototipat o *protoboards*. L'explicació es realitza pas a pas i és força entenedora.
- Finalment, Eater [6], en el seu blog, segueix també el model SAP-1 de Malvino i Brown, però des d'un altre punt de vista; ha creat un seguit de vídeos didàctics on explica el perquè de cada mòdul i cada component que ha utilitzat. A més, el muntatge és molt endreçat i visualment, molt atractiu.

Finalment, he decidit basar el meu disseny en la guia de Ben Eater, ja que a diferència de spel3o, és molt més clara i didàctica i la construcció, molt més refinada. Tot i així, he afegit certs aspectes i components inspirats per SAP-2 per aconseguir que sigui més versàtil i fàcil de programar.

4.1.3 Disseny dels components

Abans de tot, crec que val la pena presentar l'esquema general del processador, el diagrama del qual es troba a la figura 4.1. Com es pot veure, tots els components estan estructurats al voltant del bus central de 8 bits, que permetrà comunicar-los entre sí. També hi estan marcades les senyals de control que cadascun dels components rep o genera.

Per implementar cadascun dels mòduls, utilitzaré components amb un nivell d'abstracció moderat, és a dir, no implementaré cadascun dels mòduls amb portes lògiques ni tampoc

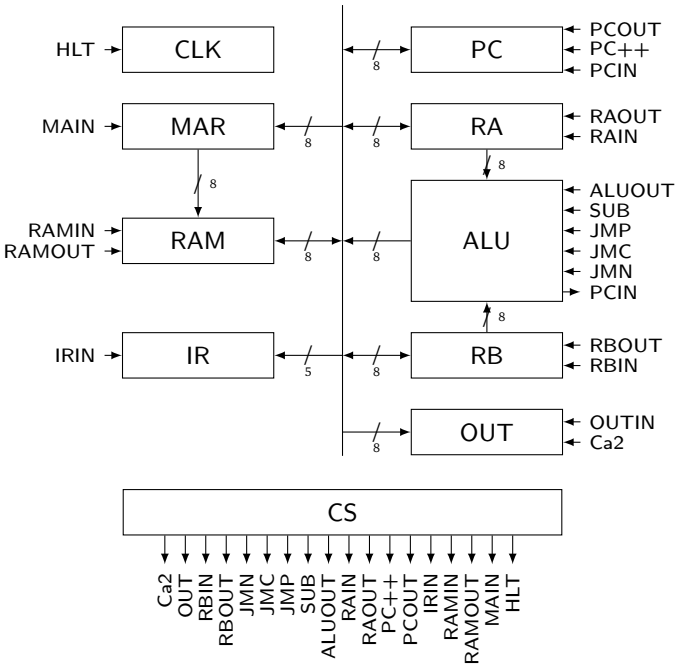

Figura 4.1: Diagrama de blocs del processador

el mòdul constarà d'un sol circuit integrat. Així, serà simple i alhora, mostrarà els diferents components essencials. Faré servir doncs els circuits integrats de la sèrie 74LS tal com recomana Malvino i Brown, comunament utilitzada en les plaques base que allotjaven els primers processadors , com l'Intel 8085. Actualment, encara se'n poden trobar.

Aquest processador l'he muntat fent servir plaques de prototipat (*protoboards*) tal com fan les dues guies que he esmentat als antecedents; permeten experimentar sense incrementar el cost de muntatge i donen un acabat final força curiós i estèticament plaent, si se segueix l'estil de muntatge d'Eater.

El processador funciona a $5V$ i té per font d'alimentació un parell de carregadors de mòbil (de $2A$). La màxima freqüència a que pot funcionar són 20Hz. Disposa de 2048 bytes de memòria RAM, però per simplicitat, només se'n poden adreçar 256. Té 2 registres de propòsit general de 8 bits que serveixen al mateix temps d'entrades pel sumador (i restador). Conté un mòdul de sortida que és capaç de mostrar un nombre de 8 bits en binari natural o Ca2 en base decimal. Com a element complementari al model original d'Eater, hi he afegit un programador automàtic, que mitjançant un microcontrolador (Arduino), puc programar molt fàcilment el processador utilitzant llenguatge assemblador.

Seguidament, explicaré les característiques més rellevants de cadascun dels mòduls. Es pot trobar informació més extensa i concreta a l'apèndix A i els esquemàtics a l'apèndix B. Més tard, comentaré la interacció entre ells amb la creació de la ISA.

4.1.3.1 Sistema de rellotge

Com he comentat anteriorment, un dels objectius del processador de 8 bits és poder mostrar el seu funcionament pas a pas. Per aquest motiu, el rellotge principal ha de tenir una velocitat variable i, preferentment, lenta, perquè es pugui apreciar la interacció entre els diferents components. També s'ha de poder generar una senyal de rellotge manualment, amb un polsador, per exemple, per facilitar l'execució pas a pas i la cerca de possibles errors durant la construcció (*debug*).

4.1.3.2 Registres

El processador té un total de 5 registres de 8 bits, dos dels quals són de propòsit general i la resta, específic. Els registres de propòsit general són el registre A (RA) i B (RB) i estan connectats directament al mòdul sumador; els de propòsit específic són el registre d'adreça de memòria (MAR), el registre d'instrucció (IR) i el registre de sortida.

Tal com menciona Malvino i Brown al seu model, el registre B només permet ser escrit. Jo he permès que pugui també ser llegit, és a dir, que pugui exposar les seves dades al bus principal, ampliant-ne la versatilitat.

4.1.3.3 Sistema de memòria

El component central del mòdul de memòria és la memòria RAM. El model que utilitzat és *CMOS Static RAM* [5], de 16Kbits de memòria, organitzada en 2048 paraules de 8 bits. Es necessiten 11 bits per adreçar la totalitat de les dades, però el bus principal que he implementat només en té 8. Per tant, per simplicitat, només adreçaré 256 posicions, que ja és una millora substancial al disseny de Malvino i Brown, on només se'n poden adreçar 16.

També forma part del sistema de memòria el registre d'adreça de memòria (MAR), que la seva funció és mantenir estable l'adreça que el la memòria RAM utilitza.

Com a complement extra, he afegit un sistema de programació que, mitjançant un Arduino i un seguit de lògica extra, permet copiar les instruccions i les dades d'un programa des d'un ordinador convencional a la memòria RAM.

4.1.3.4 Sistema aritmètic

L'espai físic de què disposo és limitat i també la quantitat de senyals de control que puc gestionar. Per aquests dos motius, vaig haver de canviar la ALU (*aritmetical logical unit*) que tenia pensat fer servir per un sumador convencional, com ho és el *4-bit Binary Full Adder With Fast Carry* [2]. Degut a que és un sumador de 4 bits, n'he utilitzat dos en paral·lel. Les entrades estaran connectades directament als registres A i B i la sortida, ho estarà al bus

principal mitjançant un *Octal Bus Transcievers With 3-State Outputs* [10]. Aquesta sortida varia quan alguna de les entrades ho fa i, per aquest motiu, el sumador no necessita cap tipus de sincronització amb el rellotge principal.

El model de Malvino i Brown també contempla l'operació resta. En binari, les restes són sumes on un dels sumands està expressat en complement a dos (Ca2). Per no complicar-ho, només permeto la representació en Ca2 del registre B (el segon sumand).

4.1.3.5 Sistema de sortida

El mòdul de sortida ha de llegir el contingut del bus de dades i mostrar-lo en base decimal usant dígits de 7 segments (com els de les calculadores). Es pot escollir la representació de les dades entre binari natural i complement a 2 (Ca2). Com que el màxim nombre representable en 8 bits és 255, faré servir 3 dígits i un quart per mostrar el signe si el nombre és negatiu. Mitjançant una memòria EEPROM he simplificat la lògica per mostrar els resultats. Per a més informació, consulteu l'apèndix A.4.

4.1.3.6 Sistema de control

El sistema de control o unitat de control s'encarrega de traduir les instruccions escrites en binari i que resideixen al registre d'instruccions en senyals de control per governar tots els elements del processador. En total, el processador que he muntat disposa de 19 senyals de control que permeten comandar els diferents mòduls perquè interactuin entre ells.

A l'apartat 4.1.4 parlaré del conjunt d'instruccions, però avanço un parell de conceptes: les instruccions que pot executar aquest maquinari tenen una llargada de 5 bits. Tot el que la instrucció especifica que farà no es pot realitzar en un sol tic de rellotge. Per això, les instruccions es divideixen en microinstruccions. He reservat 3 bits pel comptador de microinstruccions, per tant, cada instrucció en pot estar formada per 8. Així doncs, en total puc disposar de $2^8 = 256$ microinstruccions diferents.

Com en el mòdul de sortida, generar una combinació de senyals de control a partir de 256 entrades diferents comporta certa lògica que ocupa espai i augmenta la complexitat del projecte. Per tant, també faré servir memòries EEPROM per simplificar-la.

4.1.4 Disseny del conjunt d'instruccions

Tal com he introduït breument a l'apartat anterior, una microinstrucció és l'activació de certes senyals de control en un instant de temps que permetrà fer una acció al maquinari, com ara, copiar dades de memòria a un registre o incrementar el comptador de programa. Són, en general, accions molt concretes i molt lligades al maquinari.

Per simplificar la programació, existeix una abstracció de les microinstruccions, les anomenades instruccions assemblador. Aquestes, estiguin escrites en llenguatge natural o en binari, encadenaran un seguit microinstruccions per realitzar una tasca més complexa, com ara sumar un parell de nombres o guardar el contingut d'un registre a la memòria.

La mencionada tasca de la unitat de control és executar el seguit de microinstruccions corresponents a una instrucció donada.

4.1.4.1 Conjunt d'instruccions o ISA

El processador que he dissenyat suporta aquestes instruccions (escrites en assemblador):

Instrucció	Acció	Instrucció	Acció
`ADD [mem]`	`RA += *mem`	`MOVAB`	`RB = RA`
`SUB [mem]`	`RA -= *mem`	`MOVBA`	`RA = RB`
`ADDI word`	`RA += word`	`OUTU`	`printf("%u",RA)`
`SUBI word`	`RA -= word`	`OUTS`	`printf("%d",RA)`
`INC [mem]`	`*mem++`	`JMP [mem]`	`PC = mem`
`LDA [mem]`	`RA = *mem`	`JMC [mem]`	`if (carry) PC = mem`
`LDB [mem]`	`RB = *mem`	`JMN [mem]`	`if (neg) PC = mem`
`LDAI word`	`RA = word`	`HLT`	`stop the clock`
`LDBI word`	`RB = word`	`NOP`	`does nothing`
`STA word`	`*mem = RA`		
`STB word`	`*mem = RB`		

Taula 4.1: Conjunt d'instruccions del processador amb accions que realitzen. `RA` i `RB` són els dos registres de propòsit general; `*mem` representa el valor de la posició de memòria adreçada per `mem`; `word` és un nombre de 8 bits; les funcions `printf(...)` representen el mòdul de sortida; `carry` representa el bit de carry de sortida del sumador; `neg` representa el bit més significatiu de la sortida del sumador.

Una instrucció es composa de 4 etapes, generalment: lectura d'instrucció, lectura d'operands, operació i escriptura dels resultats. Així com la lectura d'operands, l'operació i l'escriptura dels resultats poden variar d'instrucció a instrucció, la lectura de la instrucció és necessària per totes les instruccions.

Lectura d'instrucció o *instruction fetch* En aquesta etapa es llegeix la instrucció per saber què s'ha de fer. El primer pas consisteix en portar el continut del comptador de programa al registre d'adreça. El següent, és carregar al registre d'instrucció el contingut de la memòria a on apunta el registre d'adreça. Finalment, incrementem en 1 el comptador de programa. En aquest moment, la unitat de control ha rebut una nova instrucció i ja sap quines microinstruccions haurà d'executar.

Lectura d'operands o *operand fetch* Totes les instruccions excepte les de moviment, les de sortida i `HLT` tenen operands. Així doncs, en aquesta tapa es llegeixen els operands de

memòria. Per no limitar la mida dels operands a 3 bits (recordo que les instruccions tenen una llargada de 5 bits), l'operand es troba a la següent posició de memòria.

Operació En aquest processador, l'única operació com a tal és una suma o una resta. De la manera que està connectat el sumador amb els registres A i B, la suma (o la resta) és immediata i constant en el temps, és a dir, que el sumador no necessita estar sincronitzada amb el rellotge. Per això, el pas d'«operació» queda ambigu en aquesta ISA.

Escriptura de resultats L'escriptura de resultats consisteix en guardar els resultats en un medi temporal per continuar l'execució. Pot ser a un registre de propòsit general, tal com es realitza, per exemple, en l'operació `ADD` o bé a la memòria RAM, com és el cas de la instrucció `STA`.

4.1.4.2 Senyals de control

Cada microinstrucció està formada per un conjunt de senyals de control. Per simplificar la comprensió de les senyals de control, em cenyiré a una convenció: 0 significa desactivada o que no té cap efecte i 1 significa activada o que canvia el comportament per defecte d'un mòdul. Tot i així, alguns dels mòduls prenen per senyal activa una senyal que és 0, altrament anomenat, *active low*. La solució més senzilla és negar aquestes senyals abans d'entrar al mòdul. A l'apèndix A.5 hi ha l'explicació detallada de la funció de cada senyal de control.

A l'apèndix A.6 hi ha la descripció completa de les senyals de control que formen cada microinstrucció de cadascuna de les instruccions que he anomenat a l'apartat 4.1.4.1. Per introduir el funcionament de les instruccions em basaré en la instrucció `ADD`, les microinstruccions de la qual estan a la taula 4.2.

Pas	Senyals de control		
1	PCOUT	MAIN	
2	RAMOUT	IRIN	PCINC
3	PCOUT	MAIN	
4	RAMOUT	MAIN	PCINC
5	RAMOUT	RBIN	
6	ALUOUT	RAIN	RST

Taula 4.2: Microinstruccions per la instrucció ADD

Els passos 1 i 2 conformen la lectura de la instrucció. Els passos 3, 4 i 5, la lectura dels operands i el pas 6, l'escriptura del resultat. L'estat d'operació és implícit, ja que es realitza al canviar els valors dels registres A i B. Ara, microinstrucció per microinstrucció, n'explicaré el significat (el número del pas coincideix amb l'enumeració següent):

1. Lectura de la posició de memòria on es trobarà la instrucció. S'ha de tenir en compte que no se sap quina instrucció és fins que no s'ha realitzat el pas 2.

2. Un cop tenim la instrucció, la passem al registre d'instrucció, En aquest moment, la unitat de control ja sap quines senyals de control segueixen per realitzar la instrucció `ADD`. També aprofitem per incrementar en 1 el comptador de programa.

3. Com he explicat anteriorment, la instrucció ADD té un paràmetre i és una posició de memòria. En aquest pas, estem llegint la posició de memòria a on haurem d'anar a buscar les dades.

4. Obtinguda la posició, la col·loquem al registre d'adreça i obtindrem la dada. Aprofitem ara per incrementar en 1 el comptador de programa.

5. Obtinguda la dada, la col·loquem al registre B per realitzar la suma. Recordo que la instrucció `ADD` realitza la suma amb el què hi hagi al registre A, per tant, prèviament haurem hagut de col·locar dades al registre A, amb la instrucció `LDA` o `LDAI`, per exemple.

6. Finalment, realitzada la suma, guardem el resultat en el registre A i activem la senyal de control interna (`RST`) que indica que aquest és el final d'instrucció. A diferència del disseny de Malvino, on totes les instruccions tenen un nombre fix de microinstruccions, jo he afegit aquesta optimització per tenir un nombre variable de microinstruccions (màxim 8), fent l'execució dels programes més eficient.

4.1.5 Funcionament general

En aquest apartat, explicaré com funciona, a alt nivell, el processador des de dos punts de vista: la posada en marxa i la sincronització interna.

El primer que cal fer quan es connecta a la corrent, és canviar el rellotge de mode automàtic a mode manual, per no interferir en la programació. Cal realitzar tantes pulsacions de rellotge com siguin necessàries (màxim 7) per posar el comptador de microinstruccions a 0, ja que per la seva construcció no es pot posar a 0 automàticament. També cal pitjar el polsador que realitza un reset a tots els components.

Un cop fet, es pot engegar el programador, que prendrà el control momentani del mòdul de memòria per escriure a la memòria RAM les instruccions i dades que prèviament s'han programat. Quan acabi, retornarà el control del mòdul de memòria a la unitat de control del processador.

En aquest moment, ja es pot començar a executar el programa, ja sigui amb el rellotge en mode manual o en mode automàtic.

Per defecte, el registre d'instruccions està a 0. Aquesta instrucció està reservada pel moment d'encendre el processador. Conté només dues microinstruccions, que són les corresponents a l'etapa de lectura d'instrucció. Un cop llegida, el registre d'instrucció canvia de valor, fent que seguidament s'executin les microinstruccions que aquest indiqui.

A partir d'aquest punt, l'execució del programa només depèn de les instruccions que el programador hagi escrit. Fins que no s'executi la instrucció `HTL`, el rellotge no quedarà aturat. No hi ha cap control sobre instruccions invàlides, és a dir, que si el registre d'instruccions conté un codi d'instrucció que no és vàlid, el comportament pot ser inesperat i possiblement causar estralls a nivell electrònic.

A nivell de sincronització, el rellotge presenta dos canvis: de 0 a 1 (*rising edge*) i de 1 a 0 (*falling edge*). Tots els components que requereixen sincronització actuen en *rising edge*. La unitat de control, però, actua en *falling edge*. Quan el rellotge passa de 1 a 0, la unitat de control canvia les senyals de control; quan el rellotge passa de 0 a 1, els components «actuen».

4.1.5.1 Codi de colors

Per tal de visualitzar el què està passant a dins el processador i distingir-ne les diferents parts, vaig afegir-hi uns leds i vaig mantenir un codi de colors pel cablejat, el significat dels quals els explicaré en aquest apartat. Abans de començar, però, vull fer un incís en la posició dels leds.

Tots els registres, excepte el de sortida, disposen de 8 leds corresponents als 8 bits que emmagatzema. El comptador de programa, tot i no ser explícitament un registre, també té leds. L'ordre: a la dreta, el bit menys significatiu i a l'esquerra, el més significatiu. També he col·locat leds a la sortida de la memòria RAM, que mostra en tot moment el contingut de la posició que indica el registre d'adreça de memòria. També hi ha tres leds que mostren el comptador d'instruccions (en aquest cas, amb ordre invers per simplificar-ne la connexió). El sumador també disposa de 8 leds amb el mateix ordre que els registres, i mostra contínuament el resultat de la suma (o resta) del registre A i B. Finalment, cada senyal de control disposa del seu led, que estarà encès quan la senyal està activada, independentment si la senyal és *active low* o *active high*.

La funció dels leds és informar sobre quin valor hi ha emmagatzemat a cada mòdul, ja que un dels objectius és ser didàctic, i per ser-ho, cal mostrar informació visual extra.

Passant al codi de colors, és força senzill: els leds grocs signifiquen senyals de control. Els leds vermells signifiquen adreces de memòria (per això, el comptador de programa i el registre d'adreça de memòria ho són). I els leds verds, signifiquen dades. Tot i que el contingut de la memòria RAM pugui ser una adreça, no deixa de ser una dada emmagatzemada, per això és de color verd.

Pel que fa a les connexions, els fils grocs són senyals de control. Els fils blaus són connexions al bus central, els fils blancs porten la senyal de rellotge i els fils verds formen la resta de connexions. També hi ha fils marrons en certs mòduls; el significat és el mateix que el fil verd, però són connexions que abasten més d'una placa de prototipat. Finalment, els fils vermells i negres formen part de la xarxa de distribució de corrent.

L'ús de diferents colors simplifica la comprensió i facilita la construcció del processador i la cerca d'errors de connexió.

4.1.6 Proves i resultats

En aquest apartat exposaré un programa real executant-se al processador per comprovar-ne el bon funcionament.

També explicaré el que es veu quan s'executa el processador, a través d'un conjunt breu de microinstruccions. I finalment comentaré els resultats obtinguts en una xarrada realitzada en un institut parlant d'aquest processador.

4.1.6.1 Sèrie de Fibonacci

Per comprovar el bon funcionament del processador, he programat la seqüència de Fibonacci. La seqüència de Fibonacci segueix aquest patró: $x_i = x_{i-1} + x_{i-2}$; $x_0 = 1$ i $x_1 = 1$; $i \geq 0$. Aquesta sèrie hauria de resultar sent: 1, 1, 2, 3, 5, 8, 13, etc. Per evitar aturar el programa quan el resultat supera 255, faig que torni a començar. La figura 4.2 representa la sèrie de Fibonacci escrita en pseudo-assemblador.

Degut a l'absència d'un assemblador, em veig obligat a escriure els programes amb el format de la figura 4.3 per poder-los copiar al processador mitjançant l'Arduino.

4.1.6.2 Problemes coneguts

Finalment, en aquest apartat m'agradaria comentar breument alguns dels problemes i mancances que m'he trobat durant el procés i que solucionaria si continués desenvolupant aquesta arquitectura:

- Inestabilitat de les connexions degut a la naturalesa de les plaques de prototipat.
- Manca de l'habilitat per comparar nombres, i per tant, manca d'instruccions de salt tant útils com `JMZ` (*Jump if 0*).
- Tal com està muntat, és difícilment escalable; afegir nous mòduls sense haver de modificar-lo gaire és pràcticament impossible.
- Manca d'un assemblador, que llegeixi codi d'un fitxer de text i l'interpreti.

```
.main:
0.  LDAI 0
2.  STA 251
4.  LDAI 1
6.  STA 250

8.  ADD 251
10. OUTU
11. STA 252

13. LDA 250
15. STA 251

17. LDA 252
19. STA 250

21. JMC 0
23. JMP 8

.data:
250. 0
251. 0
252. 0
```

Figura 4.2: Sèrie de Fibonacci escria en llenguatge pseudo-assemblador per l'arquitectura que he dissenyat. Els números de línia corresponen a les posicions de memòria. Hi ha instruccions que ocupen dues posicions (una per la instrucció i l'altre pel paràmetre) i d'altres que només n'ocupen una.

```
static instruction fibonacci[13] =
{
    {LDAI, 0},
    {STA, 251},
    {LDAI, 1},
    {STA, 250},

    {ADD, 251},
    {OUTU, 0},
    {STA, 252},

    {LDA, 250},
    {STA, 251},

    {LDA, 252},
    {STA, 250},

    {JMC, 0},
    {JMP, 8}
};
static data fibonacci_data[3] =
{
    {250, 0},
    {251, 0},
    {252, 0}
};
```

Figura 4.3: Sèrie de Fibonacci en el format per poder programar el processador. Aquesta estructura és llegida per l'Arduino per col·locar les instruccions i les dades al lloc corresponent de la memòria RAM.

4.1.7 Conclusions dels resultats

Dissenyar un processador que compleixi els objectius marcats comporta alguns reptes; dissenyar-lo per muntar-lo a mà, en comporta de més complicats, ja que s'ha de contenir el nombre de components i les connexions sota un límit. Si a tot això afegeixo el repte que el resultat final sigui visualment plaent i instructiu, fa que tant el disseny com el muntatge hagi portat certa feina. En certa manera, el disseny de processadors comercials han de tenir en compte que la quantitat de components està limitada per l'àrea del material semiconductor i que la complexitat de les connexions ha d'estar controlada.

El mòdul que he trobat més complicat de dissenyar ha sigut el conjunt de registre de memòria i memòria RAM, ja que el fet d'afegir un programador ha incrementat el nombre de components i la complexitat de les connexions.

Si hagués de dir quin mòdul ha sigut més complicat de muntar, diria que cap en concret. Un cop tenia el disseny sobre paper, calia plantejar la disposició dels components i les connexions. El que ha portat més temps és tallar cadascun dels fils (més de 9 metres de fil) i fer que quedin plans.

El mòdul que ocupa més espai, sens dubte, és la unitat de control, tot i ser relativament simple. Les tres memòries EEPROM i els 19 leds fan que aquest mòdul s'estengui a voi 3 plaques. A més, per la seva naturalesa, té connexions que recorren tota la superfície del processador portant les senyals de control a cadascun dels mòduls.

Aprofito el moment per mencionar la solució als problemes esmentats a l'apartat 4.1.6.2, com a feina futura pel desenvolupament de la nova versió. Als tres primers problemes, la solució més pràctica és dissenyar el circuit en plaques de circuit imprès (de doble capa); s'elimina el problema de connexions inestables i permeten afegir més components en menys espai, podent afegir una ALU completa i podent afegir connexions extres per nous mòduls. La solució per la manca d'un assemblador és clara: cal definir el llenguatge i programar-lo.

També m'agradaria fer esmena de les diferències, a grans trets, és clar, que té el processador que he dissenyat amb els processadors actuals, ja siguin de l'estil CPU o GPU (els dos tipus d'arquitectures més conegudes, actualment).

Els processadors de propòsit general o CPU, a l'igual que el processador que he dissenyat, tenen un sistema d'accés a la memòria, registres de propòsit general i específic, una o més ALUs (sumador en el meu cas) per realitzar operacions i una unitat de control. Però també disposen d'altres tècniques per accelerar els càlculs, com ara:

- *Pipelines* d'execució, que permeten paral·lelitzar les diferents etapes de la instrucció (lectura d'instrucció i d'operands, operació i escriptura de resultats) i, d'aquesta manera, acostar-se a l'ideal d'una instrucció per cicle de rellotge (IPC – *Instructions Per Clock* $\rightarrow 1$).
- Prediccions de salt, que permeten mantenir el *pipeline* constantment ple.
- Memòries cau coherents, repartides en diferents nivells i en els diferents nuclis del processador, que permeten una reducció de la latència d'accés a memòria i una millora de l'ample de banda efectiu, sense complicar (gaire) la vida del programador.
- Unitats de càlcul específiques, entre d'altres.

Totes aquestes tècniques són molt complexes i estan implementades directament al maquinari, perquè siguin ràpides i eficients, fent que la superfície total d'un nucli del processador sigui força gran. Per aquest motiu, com que la superfície total és limitada pel cost de construcció, el consum energètic i la dissipació de la calor, no se'n poden posar gaires (actualment, d'entre 4 i 8 en les configuracions més habituals).

Així doncs, els processadors de propòsit general són molt eficients en executar tasques de caràcter serial i són molt eficients en gestionar salts de programa (*branching*) i en tractar grans conjunts de memòria dispersa. Però, degut a les poques unitats de càlcul, fallen en problemes de paral·lelisme de dades, és a dir, en problemes on s'ha d'aplicar una mateixa operació a un gran conjunt de dades.

El processador que he muntat només disposa d'un nucli; si hagués de fer que tingués més nuclis, hauria de duplicar moltes parts del processador, com ara els registres de propòsit general, la ALU, el comptador de programa, etc i ampliar enormement la unitat de control perquè pogués gestionar els accessos als recursos compartits, com ara el bus central o la memòria RAM. Si hagués d'implementar un sistema de *pipeline* o de memòria cau, la superfície i la complexitat augmentarien enormement. Seria més flexible i més ràpid, sí, però necessitaria molt més espai i seria més complicat d'entendre el funcionament.

Pel que fa a les targetes gràfiques o GPU (*Graphic Processing Unit*), van ser originalment dissenyades pel càlcul gràfic eficient, és a dir, per un propòsit específic. Tot i així, ja fa uns anys que ha sorgit el concepte GPGPU (*General Purpose GPU*), és a dir, utilitzar aquest maquinari per tasques més generals que requereixin gran paral·lelisme de dades. Una GPU pot arribar a tenir més de 5000 nuclis, contrastant amb la quantitat de nuclis d'una CPU. Aquest fet és gràcies a que:

- Els nuclis són molt senzills, i, per tant, petits.
- La memòria cau és molt petita i menys complexa.
- Però es veu compensat per la presència de memòries locals explícitament programables. Això desplaça la complexitat del maquinari (deixant més superfície utilitzable per altres nuclis) al programari. Aquest fet redueix els costs de producció però encareix els costs de desenvolupament per aquest tipus de maquinari.
- L'execució de diferents nuclis explícitament paral·lela, és a dir, que cert nombre de nuclis compartiran el mateix comptador de programa, emulant així l'ús vectorial dels processadors.

Tornant a l'analogia amb el meu processador, si l'hagués de transformar perquè s'assemblés a l'arquitectura d'una targeta gràfica, hauria d'afegir més sumadors, 3 més, per exemple, incrementant també el nombre de registres.

Això ocuparia moltíssim menys que tenir 4 nuclis de CPU i faria que el processador fos més útil i econòmic per a aplicacions que tinguin un alt paral·lelisme de dades, però seria menys flexible. De totes maneres, gestionar instruccions vectorials pot simplificar la unitat de control, aprofitant la coalescència de les dades i fer que una sola senyal activi un conjunt de canals de dades per a n ALUs.

Finalment, m'agradaria tornar al començament de la secció i repassar si he complert els objectius que he proposat:

He dissenyat un processador funcional i, que a més, és *turing complet*[1]. L'he pogut muntar i hi he executat programes satisfactòriament. També ha servit com a element didàctic, ja que en el moment d'escriure aquesta memòria, ja he fet una xarrada en un institut mostrant aquest processador als alumnes de batxillerat.

També he aprofitat aquesta tasca per entendre millor les decisions que s'han de realitzar a nivell de disseny en les diferents arquitectures comercials, com ara, quantitat de components o dificultat de connexió entre ells.

[1]Un sistema lògic o un llenguatge de programació es considera TURING COMPLET si es pot equiparar a una màquina universal de Turing.

4.2 Programació conscient de l'arquitectura aplicada

En aquesta secció, analitzaré un problema de matemàtica discreta. L'objectiu serà mostrar diferents maneres de resoldre'l utilitzant diferents arquitectures comercials, aplicant el concepte de programació conscient de l'arquitectura. Aquesta implementació serà purament acadèmica, és a dir, sense buscar altres finalitats que no sigui l'optimització. Començaré presentant l'algorisme bàsic amb optimitzacions pròpies, i acabaré detallant tres implementacions que utilitzen parts específiques del maquinari dissenyades per accelerar càlculs matemàtics.

4.2.1 Motivació

A la secció anterior he creat una arquitectura, capaç d'executar codi molt senzill, però incapaç d'executar programes més complexos, degut a la simplicitat del disseny. En aquesta secció, doncs, faré ús d'arquitectures i llenguatges comercials per tal d'explorar la programació conscient de l'arquitectura (PCA).

Utilitzaré processadors de propòsit general (CPU), que són el nucli de tots els ordinadors personals d'avui en dia i també targetes gràfiques (GPU), cada cop més freqüents per la gran capacitat computacional que disposen. Així podré veure des d'un llenguatge d'alt nivell, com C, com treballar el concepte de programació conscient de l'arquitectura i profunditzar-hi, ja que cadascuna d'aquestes arquitectures requereix d'un mínim coneixement del seu funcionament per entendre quina és la manera més eficient de programar-les.

Hi ha algorismes que funcionen bé en CPU, com és el cas de la cerca de nombres primers. Amb aquests algorismes, vull veure què puc fer per poder-lo executar eficientment a la GPU; vull entendre perquè es diu que un algorisme funciona millor en una certa arquitectura i pitjor en l'altre; vull observar els problemes que podré trobar. Així adquiriré una comprensió més profunda de quins components del maquinari fan que un algorisme determinat sigui més ràpid o més lent, i quins es beneficien més o menys de certs patrons d'accés a memòria dels algorismes.

4.2.2 Antecedents

A la motivació he mencionat dues arquitectures comercials que es poden trobar en la majoria dels ordinadors actuals (per no dir tots): CPU i GPU. En aquest apartat, breument, exposaré un seguit de programes o algorismes que les fan servir amb la idea de la programació conscient de l'arquitectura molt present:

Prime95 és una aplicació que es dedica a la recerca de nombres primers de Mersenne i ho fa exclusivament utilitzant la CPU. Existeixen també aplicacions que creen nombres de coma flotant d'una precisió increïble per calcular decimals de nombres irracionals, com els de π

o els de e. Molts altres utilitzen la CPU per solucionar problemes de matemàtica complexa, que per la seva naturalesa, és una arquitectura idònia.

Pel que fa a l'ús conscient de l'arquitectura de GPU, els programes de disseny i modelatge en 3 dimensions, com ara Blender, utilitza la targeta gràfica per mostrar el model per la pantalla o renderitzar el resultat final. Programes científics utilitzen la GPU per realitzar simulacions de partícules, sintetització de proteïnes, prediccions meteorològiques, etc, que són algorismes que es basen en la paral·lelització de dades.

Finalment, vull introduir un concepte anomenat *Collaborative Computing*, és a dir, computació col·laborativa, que es basa en la interoperabilitat entre diferents arquitectures, comunament, CPU i GPU. Es tracta de trencar un problema computacionalment intens en diferents parts perquè puguin ser executades en paral·lel en diferents arquitectures i optimitzar cada part per una arquitectura específica. Un exemple d'aplicació d'aquest estil de programació són els programes de processat d'imatge en temps real, com ara els que utilitzen vehicles autònoms, càmeres de supervisió en una cadena de muntatge, video-vigilància, etc.

Per dur a terme l'exploració de la programació conscient de l'arquitectura, he escollit una tipologia d'algorisme que funciona bé amb CPU, per descobrir quines són les dificultats que descarten l'ús del paral·lelisme de GPU i perquè. He escollit la cerca de nombres primers petits (menors que 2^{64}). Existeixen diferents algorismes de cerca, però potser els més coneguts són:

- **Sedàs d'Eratòstenes**. És possiblement l'algorisme de cerca de nombres primers petits més conegut, gràcies a la seva senzillesa i a la fàcil implementació. Hi ha nombroses implementacions d'aquest algorisme, però Walisch [15] assegura que n'és la més ràpida.
- **Sedàs d'Atkin**. Matemàticament, aquest sedàs està més optimitzat que el d'Eratòstenes. També hi ha nombroses implementacions, però la més coneguda és la de Bernstein [4].
- Test de primeritat d'**AKS** (Agrawal–Kayal–Saxena). Aquest test és general, de temps polinòmic, determinista i no condicional; en altres paraules: funciona per tot nombre d'entrada, el temps de còmput es pot calcular a partir del nombre de xifres de l'entrada, els resultats no estan basats en models probabilístics i no fa ús de cap hipòtesi prèviament no provada. Aquest algorisme és molt eficient per nombres grans, però la base matemàtica és molt densa.

4.2.3 El sedàs d'Eratòstenes

He escollit el sedàs d'Eratòstenes per la senzillesa de l'algorisme i perquè és fàcilment modularitzable; així em permetrà separar les parts més paral·lelitzables de les menys, podent

decidir en quina arquitectura executar-les i poder fer ús de la tècnica de *collaborative computing*.

El sedàs d'Eratòstenes és un algorisme capaç de trobar tots els nombres primers menors que un límit preestablert. Ho fa iterativament, marcant com a nombre compost els múltiples dels nombres que no han sigut prèviament marcats, començant pel número 2. A l'acabar, tots els nombres que no han quedat marcats, són els nombres primers.

Els passos de l'algorisme són els següents:

1. Genera una llista de nombres des de 2 fins a n: $\{2, ..., n\}$.
2. Inicialment, defineix $p = 2$, sent p el primer més petit.
3. Començant per $2p$ i en passos de p fins a n, marca tots els nombres (els múltiples de p).
4. Començant per l'inici de la llista, cerca el primer nombre $m > p$ que no hagi sigut marcat. Si no en queda cap, salta al pas 6.
5. Assigna $p = m$ i torna al pas 3.
6. Tots els nombres no marcats són nombres primers.

Analitzant amb deteniment aquest algorisme, podem observar que en el pas 3 es pot començar per p^2 enlloc de $2p$, ja que ja s'han comprovat tots els múltiples de p menors que p. També es pot observar que la condició de parada es pot canviar per $p^2 > n$, per dues raons equivalents:

- La primera, per la optimització esmentada anteriorment; si comencem per p^2 i $p^2 > n$, no queden més nombres per comprovar.
- I la segona, si un nombre x no és primer, llavors es pot expressar com $x = a \cdot b$, on a i b són nombres naturals positius. Si a i b són més grans que $\sqrt{x}$, llavors el seu producte seria més gran que x. Per tant, un dels dos nombres ha de ser menor que $\sqrt{x}$ (o els dos ser $\sqrt{x}$). Com que el producte de nombres naturals és commutatiu, comprovant tots els nombres menors que l'arrel de x és suficient. La condició de parada $p^2 > x$ és equivalent a $p > \sqrt{x}$.

Aquestes dues optimitzacions són optimitzacions a nivell matemàtic, que acostumen a ser les que realment acceleren el càlcul computacional. Però un cop aplicades, cal observar amb deteniment el codi resultant i buscar els possibles colls d'ampolla segons l'arquitectura on s'executi aquest algorisme.

4.2.3.1 Segmentació del sedàs

Hi ha diferents maneres de transformar el sedàs d'Eratòstenes per eliminar els límits imposats: 2 i n. A continuació, presentaré l'algorisme *Segmented Sieve* [12], o segmentació del sedàs, capaç de trobar nombres primers donats dos límits qualssevol:

> Cal trobar tots els nombres primers compresos entre a i b. Per fer-ho, cal cercar prèviament els nombres primers compresos entre 2 i $\sqrt{b}$. Amb aquests, podrem discernir quins nombres d'entre a i b són primers i quins no.

Per clarificar-ho, mostro el pseudocodi corresponent a aquest algorisme:

Algorisme 1: Segmentació del sedàs

input : two positive odd integers a and b where $a < b$
output: a list of prime numbers between a and b

```
1 numbersToCheck ← list of odd consecutive numbers from a to b;
2 listOfPrimes ← list of all primes smaller than √b;
3 foreach x in numbersToCheck do
4     if x has not been marked then
5         foreach p in listOfPrimes do
6             if p | x then
7                 mark all numbers from x to b in steps of p;
8                 break;
```

4.2.3.2 Abstracció de l'algorisme

L'algorisme explicat anteriorment cerca la primera aparició d'un p i marca la resta de múltiples des d'on l'ha trobat fins a b. Així doncs, hi ha dues operacions: la cerca i la marcació. Són fàcilment separables, però la quantitat d'operacions continua sent la mateixa.

Els múltiples d'un nombre qualsevol són equidistants entre ells a la recta dels nombres naturals, per la definició de *múltiple*. Prenent un inici qualsevol, per exemple, a, es pot anotar la posició del primer múltiple d'un p qualsevol començant per a. Concretant, donat $a = 101$ i $p = 3$, la posició del primer múltiple de p (105) començant per a és 2, tenint en compte que la llista de nombres on es realitza la cerca no conté cap nombre parell i que la indexació comença per 0. La llista de posicions que genera aquest càlcul l'anomenaré *llista d'índexs*. El nombre d'operacions per crear-la és similar al de l'algorisme de segmentació explicat anteriorment. Tot i així, s'hi han de sumar les operacions necessàries per generar la llista de nombres primers a partir de la llista d'índexs.

El fet de generar aquesta llista d'índexs comporta un estalvi de càlcul en certes circumstàncies; suposem que s'ha generat aquesta llista d'índexs donats dos nombres a i b, $a < b$. Si

el següent pas és trobar nombres primers compresos entre c i d, $c < d$, $c \geq a$ i $d \leq b$, no és necessari tornar a recalcular la llista; adaptant els índexs guardats n'hi ha prou. La llargada de la llista d'índexs es correspon amb la quantitat de nombres primers menors que $\sqrt{b}$. Els passos que segueixen s'han de realitzar per cadascuna de les posicions de la llista, que es corresponen a un nombre primer p en concret:

1. Cal trobar el nombre x de passos de mida p que hi ha des del múltiple que assenyala la posició de la llista fins a c.
2. Sumant $x \times p$ a aquest múltiple, s'obté el primer múltiple m de p més gran que c.
3. El nou índex corresponent a p serà el resultat de $c - m$.

Si $d > b$, llavors la llista ja calculada no pot servir, car no conté totes les posicions corresponents als nombres primers menors que $\sqrt{d}$.

Finalment, cal considerar un cas especial que redueix exorbitantment el nombre de càlculs per crear la llista d'índexs, que és considerar $a = 1$. Cada posició de la llista resultarà ser $p/2$. Si considerem $b = 2^{64}$, que és l'enter més gran que els ordinadors actuals suporten de manera nativa, l'algorisme queda reduït a calcular tots els nombres primers menors que 2^{32} i dividir cadascun dels nombres entre 2 per crear la llista d'índexs. Només per mostrar el benefici aconseguit, dir que sense aquesta optimització, el càlcul passa de tardar setmanes a dècimes de segon.

4.2.4 Implementació

Un cop he presentat els algorismes que faré servir, toca implementar-los d'una manera específica segons l'arquitectura utilitzada. A l'apartat de resultats (4.2.5) compararé el temps emprat de cada implementació a nivell individual i a nivell global.

4.2.4.1 Multi-fil a la CPU

La programació multi-fil o concurrent consisteix en crear diferents fils d'execució per tal de repartir les tasques entre ells i realitzar-les en paral·lel[2]. El nombre de fils a crear idealment hauria de coincidir amb el nombre de nuclis que disposa el processador. Per fer-ho, utilitzaré la llibreria OpenMP, que simplifica moltíssim la tasca de creació, destrucció i repartiment de tasques entre diferents fils.

OpenMP funciona, generalment, a base de directives de pre-processador, `#pragma`, que són punts on el compilador delegarà certes tasques a aquesta llibreria perquè generi el codi assemblador corresponent. No voldria allargar-me massa en l'explicació del funcionament

[2]Per a més informació sobre la programació concurrent, consulteu l'apèndix C.1.

```
#pragma omp parallel for schedule(static)
for (size_t i = 0 ; i < sieve->size; i++) {
    listOfIndexes->data[i] = sieve->data[i] / 2;
}
```

Figura 4.4: Paral·lelització del bucle `for` per crear la llista d'índexs usant directives d'OpenMP.

d'OpenMP perquè és molt extensa i no és d'interès per aquest apartat, però sí que comentaré l'expressió d'aquest fragment de codi (figura 4.4).

Comunament, s'usen dues directives per declarar que un fragment de codi s'executi en paral·lel: `#pragma omp parallel` i `#pragma omp for`, la segona niuada dins la primera. Però en aquest cas, la secció paral·lela es conforma només d'un sol bucle, per tant, es poden congregar. He afegit un paràmetre extra a la directiva: `schedule(static)`. Aquest paràmetre reparteix en parts iguals la llista que s'està processant (tantes com nombre de nuclis tingui el processador). Un altre paràmetre comunament utilitzat és `schedule(dynamic)`, que divideix la tasca en moltes més subtasques que el nombre de nuclis; un d'ells s'encarregarà d'anar repartint les subtasques en els nuclis que hagin enllestit la subtasca anterior. Comporta més gestió, però per subtasques on el temps de càlcul no és previsible (cada iteració té un temps d'execució molt diferent), augmenta el rendiment general.

4.2.4.2 Instruccions vectorials a la CPU

Les instruccions vectorials o SIMD (*Single Instruction Multiple Data*) permeten aplicar la mateixa operació a múltiples dades al mateix instant, és a dir, que es tracta de paral·lelisme a nivell de dades, no pas de concurrència (que és el cas d'OpenMP)[3].

Els processadors que utilitzaré per realitzar les proves disposen de les extensions necessàries per executar instruccions SSE i AVX2. Per defecte, el llenguatge C no disposa de sintaxi per transformar codi en instruccions SIMD, així que si es volen utilitzar, s'ha d'incrustar codi assemblador. Per evitar-ho, la majoria dels compiladors (GCC per exemple) inclouen fitxers de capçalera amb funcions que embolcallen el llenguatge assemblador corresponent a la instrucció SIMD que es vol utilitzar. Aquestes capçaleres reben el nom d'*intrínsics*. *Inel Intrinsics Guide* [7] és una guia interactiva per treballar amb aquest tipus d'instruccions. No entraré en la sintaxi que utilitzen, ja que és molt senzilla; aniré directament a la implementació del bucle `for` per crear la llista d'índexs, ja que és una tasca senzilla que no implica divergència. El fragment de codi de la figura 4.5 ho exemplifica.

El primer pas és cercar la instrucció que convé. Es tracta de dividir un nombre entre 2, que és el mateix que desplaçar els seus bits una posició cap a la dreta. La instrucció que ho fa es diu `srli` (*Shift Right packed Integers*). Aquesta instrucció es troba en la variant

[3] Per a més informació sobre les instruccions vectorials o SIMD, consulteu l'apèndix C.2.

```
__m256i *sieve   = (__m256i *)sieve->data;
__m256i *indexes = (__m256i *)listOfIndexes->data;
for (size_t i = 0; i < sieve->size / 8; i++) {
    indexes[i] = _mm256_srli_epi32(sieve[i], 1);
}
```

Figura 4.5: Paral·lelització del bucle `for` per crear la llista d'índexs usant instruccions SIMD (AVX2 en aquest cas).

SSE2 (`_mm_srli_epi32` amb un màxim de 128 bits per registre) i en la variant AVX2 (`_mm256_srli_epi32` amb un màxim 256 bits per registre).

Un cop cercada la instrucció, s'han de reorganitzar les dades: els nombres que s'estan dividint entre 2 són de 32 bits; per tant, si s'usen instruccions SSE2 es podran calcular quatre divisions a la vegada i si s'usen instruccions AVX2, se'n podran fer vuit. Això significa que la mida del vector ha de ser múltiple de 4 o de 8 i que la memòria ha d'estar correctament alineada[4].

Finalment, cal realitzar el cast i executar el bucle for, però enlloc de recórrer tots els elements, només es recorrerà un quart o una vuitena part part del vector, segons si es fa servir la instrucció SSE2 o AVX2, respectivament.

4.2.4.3 GPU

En aquest apartat, implementaré part de l'algorisme per l'arquitectura de GPU, i per fer-ho, utilitzaré OpenCL (*Open Computing Language*), un estàndard obert i multiplataforma que permet programar tot tipus de sistemes heterogenis, com per exemple, un ordinador amb una targeta gràfica[5]. A més, crearé una modalitat d'execució que fa ús de *collaborative computing* per realitzar cadascuna de les tasques de què es composa el programa a l'arquitectura més adequada, paral·lelament i asíncronament.

Els algorismes que són fàcilment adaptables per ser executat en targetes gràfiques són aquells que realitzen un seguit de càlculs sobre un element del vector d'entrada i el guarden a la mateixa posició del vector de sortida. Dels diferents algorismes que composen el programa per crear primers, el que crea i el que tradueix la llista d'índexs s'adeqüen perfectament a aquestes característiques. A les figures 4.6 i 4.7 hi ha una comparació de l'algorisme de creació de primers realitzat a la CPU i l'altre realitzar a la GPU, respectivament.

[4]Una porció de memòria reservada que estigui alineada significa que l'adreça del primer element ha de ser divisible per la mida de l'element en sí. En altres paraules, si tinc un vector d'enters de 32 bits cadascun, l'adreça del primer nombre ha de ser divisible entre 4; si tinc un vector d'elements de n bits cadascun, l'adreça del primer element ha de ser divisible entre $n/8$. Generalment, la funció `malloc(...)` proporciona adreces alineades per elements de 64 bits, que també estan alineades per elements de 32, 16 i 8 bits. Per obtenir altres tipus d'alineament, és necessari utilitzar la funció `_mm_malloc(...)` dels intrínsecs, ja que pren per argument el tipus d'alineament que s'ha de complir.

[5]Per a més informació sobre OpenCL, consulteu l'apèndix C.3.

```
void create_index_list(
    List *listOfIndexes,
    List *sieve)
{
    for (size_t i = 0; i < indexListData->sieve->size; i++) {
        listOfIndexes->data[i] = sieve->data[i] >> 1;
    }
}
```

Figura 4.6: Versió de l'algorisme de creació de la llista d'índexs per CPU.

```
__kernel void create_indexList(
    __global unsigned int *prime_list,
    __global unsigned long int *length,
    __global unsigned int *indexList)
{
    unsigned long int i = get_global_id(0);
    if (i < *length)
        indexList[i] = prime_list[i] >> 1;
}
```

Figura 4.7: Versió de l'algorisme de creació de la llista d'índexs per GPU.

Deixant de banda les declaracions de les funcions i l'accés a les estructures internes de dades, la diferència principal és l'absència del bucle for a la versió de la GPU. Això és degut a la naturalesa de la seva arquitectura: quan s'enfila una tasca a la targeta gràfica, s'ha d'indicar de quants elements està formada (en aquest cas, la mida del vector). La unitat de control de la GPU, l'*scheduler*, dividirà aquest problema en unitats simples i el repartirà entre tots els seus nuclis. Una consideració a tenir en compte és la propietat que ha de tenir el nombre de tasques a realitzar perquè el càlcul sigui eficient; si aquest nombre no és potència de 2, llavors l'*scheduler* no en farà un bon repartiment. Per posar-ho en números, un dels càlculs pot passar de tardar 80 ms a només 3 utilitzant una mida de vector potència de 2. Aquest és el motiu de la presencia de l'`if`: les tasques que quedin fora del rang del vector, no es realitzaran.

Com he mencionat anteriorment, els nuclis de la GPU són molt senzills. S'agrupen en grups de 64 o més en el que s'anomena *Compute Unit* en terminologia OpenCL o *Stream Multiprocessor* en terminologia CUDA (*framework* semblant a OpenCL propietari d'NVIDIA). Tots aquests comparteixen una mateixa unitat de control, el banc de registres, el banc de memòria local, el primer nivell de memòria cau i un número concret de línies d'accés a la memòria global.

Un *warp* és un conjunt de fils de GPU agrupats de 32 en 32 o de 64 en 64. Aquesta agrupació es podria equiparar amb un fil de CPU que executa instruccions vectorials de 32 o 64 elements de 32 bits. Les instruccions assemblador de les GPU són instruccions que operen en 32 o 64 elements a la vegada i que estan executades de manera seqüencial per un *warp*.

Tot i així, els *warps* poden gestionar divergències desactivant els nuclis que no compleixen la condició i reactivant-los quan es compleix. És a dir, que si un bloc de codi consta de l'estructura `if else` , tots els nuclis del *warp* executaran tant la secció de l'`if` com la de l'`else`, però només aquells que compleixin la condició produiran resultats. El resultat final és l'execució de més instruccions, fet que disminueix el rendiment. És per aquest motiu que la introducció d'`if`s i altres elements que provoquin divergència causa problemes de rendiment. Alguns d'aquests casos es poden solucionar amb operacions a nivell de bit.

En l'algorisme de traducció, hi ha certa divergència; alguns casos els he solucionat fent operacions a nivell de bit. Tot i així, el codi divergent és molt petit i no perjudica tant al rendiment.

Finalment, vull mencionar una de les tècniques de programació conscient de l'arquitectura que he anomenat als antecedents: *collaborative computing*. Aquesta tècnica es basa en combinar les habilitats de la CPU i de la GPU per resoldre un problema de manera conjunta. Bé, l'algorisme que he plantejat per trobar nombres primers consta de dues parts principals: la creació i traducció de la llista d'índexs i la generació dels nombres primers a partir d'aquesta llista. Com he mostrat, la gestió de la llista d'índexs es pot fer a la targeta gràfica, ja que la naturalesa de l'algorisme encaixa amb la filosofia de les GPU. En canvi, l'extracció dels primers a partir de la llista d'índexs és un algorisme que recórrer una llista múltiples vegades, modificant-ne valors. Aquesta llista acostuma a ser molt gran i a cada iteració, els accessos estan més allunyats entre sí.

Les memòries cau de la GPU són, en relació al nombre de nuclis, molt més petites que les de la CPU. Com més nuclis treballin en la mateixa porció de memòria, hi ha més possibilitats de *caché miss*. Per aquest motiu, la gestió extra de moviment de dades en l'estructura de memòria de la GPU penalitza molt més que la quantitat de nuclis extra que posseeix, en relació a la CPU, que té unes memòries cau molt més grans i menys nuclis.

Amb l'objectiu d'aconseguir que la CPU i la GPU treballin en paral·lel, he creat una cua asíncrona, concurrent i circular que funciona de la següent manera: el fil principal, abans de tot, crearà *n* consumidors de CPU. Després, anirà enfilant tasques a la targeta gràfica fins a omplir la cua i s'esperarà a que es buidi per continuar enfilant. D'altra banda, els consumidors esperaran a que la targeta gràfica finalitzi la tasca; quan acabi, n'agafaran una i crearan la llista de primers. Un cop acabat, es marcarà la tasca com acabada i el fil principal podrà enfilar-ne una altra. Aquest mecanisme és complicat perquè utilitza búfers de memòria reservats a la inicialització i s'ha d'anar en molt de compte que no se sobreescriguin mentre s'estiguin llegint. Ho faig d'aquesta manera perquè la reserva de memòria (tant a la GPU com a la CPU) és costosa.

4.2.5 Proves i resultats

En aquest apartat executaré el programa creat en diferents modalitats, analitzant tant el rendiment d'alguns algorismes en concret com el rendiment global. El programa cercarà els nombres primers compresos en un interval, expressat amb l'inici i la seva mida. Si s'especifica l'opció de múltiples iteracions, el programa cercarà nombres primers en tants intervals consecutius com s'hagin especificat.

Els passos algorísmics per aconseguir aquest objectiu són els següents:

1. És necessari crear una llista (el sedàs) amb tots els primers menors que l'arrel quadrada d'un màxim pre-establert. Per defecte, aquest màxim és $2^{58} = 288230376151711744$. L'algorisme que calcula aquesta llista és seqüencial i difícilment paral·lelitzable. Només s'executa un sol cop a l'iniciar el programa.
2. Creació de la llista d'índexs a partir del sedàs.
3. Traducció de la llista d'índexs.
4. Generació dels nombres primers dins el rang especificat a partir de la llista d'índexs traduïda i del sedàs.

Hi ha dues modalitats bàsiques d'execució: la iterativa i la que fa ús de la computació col·laborativa; la iterativa executarà els tres últims algorismes de forma seqüencial tantes vegades com el nombre d'iteracions especifiqui. El mode de computació col·laborativa té el mateix objectiu, però executarà els tres últims algorismes de manera concurrent i en arquitectures diferents.

Els tests que realitzaré seran a nivell individual per cadascun dels tres últims algorismes i a nivell global.

Si no s'indica el contrari, tots els tests estaran realitzats amb aquest maquinari: Intel Core i7 4790 @ 4 GHz, 8 GB de RAM i la targeta gràfica NVIDIA GTX 750 Ti. El sistema operatiu utilitzat és Ubuntu Linux 16.04.

4.2.5.1 Tests individuals dels passos de l'algorisme

Test de creació de la llista d'índexs

D'aquest algorisme, en disposo 4 versions: utilitzant un fil d'execució, amb paral·lelisme de fil (OpenMP), amb instruccions vectorials i amb la targeta gràfica.

El nombre d'iteracions, per defecte és 1. Per tenir uns valors més significatius, executaré aquest test 4 vegades i calcularé la mitjana del temps emprat per crear la llista d'índexs, per

cadascun dels 4 mètodes. Degut a que el compilador realitza tasques d'optimització a més baix nivell, consideraré dues versions del test: sense cap optimització del compilador i amb totes les optimitzacions del compilador activades.

Sense optimitzacions Quan s'afegeix l'opció `-g`, el compilador no realitza cap optimització. Aquesta opció també pren el nom de *debug*. Els resultats es troben a la taula 4.3. Per seleccionar el mètode d'execució, s'ha d'afegir l'argument de la primera columna amb el valor corresponent.

Paràmetre	Creació de la llista d'índexs (ms)
`--iplatform=cpu`	70.58
`--iplatform=cpu -t`	25.31
`--iplatform=simd`	19.54
`--iplatform=gpu`	**4.18**

Taula 4.3: Temps per a la creació de la llista d'índexs usant els 4 mètodes d'execució. La llista té una llargada de 28.192.749 elements.

El processador que he utilitzat per fer les proves té 4 nuclis amb la tecnologia *Hyper-Threading*, fent que el nombre total de nuclis vists pel sistema operatiu siguin 8. Tot i així, amb el segon mètode, que s'executa fent servir tots els nuclis de la màquina, no es veu una reducció del temps 8 vegades menor.

Amb optimitzacions Quan s'afegeix l'opció `-Ox`, el compilador realitza un seguit d'optimitzacions que es poden trobar en aquesta pàgina: *Options That Control Optimization* [11]. Existeixen diferents nivells d'optimització: `-O -O1`, `-O2`, `-O3`, `-O0`, `-Os`, `-Ofast` i `-Og`. L'explicació de cadascun d'aquests modes es pot trobar *onine*. Per defecte, CMake, quan genera el projecte en mode *Release*, afegeix el nivell d'optimització màxim, `-O3`. Mostro ara els resultats que he obtingut a la taula 4.4.

Paràmetre	Creació de la llista d'índexs (ms)
`-iplatform=cpu`	16.82
`-iplatform=cpu -t`	21.91
`-iplatform=simd`	16.41
`-iplatform=gpu`	**4.18**

Taula 4.4: Temps per a la creació de la llista d'índexs afegint les optimitzacions del compilador. La llista també té una llargada de 28.192.749 elements.

La reducció de temps és força considerable respecte el test anterior. Fins i tot es pot observar que fer servir tots els nuclis de la màquina acaba resultant ser pitjor que només usar-ne 1, en aquest tipus d'algorisme, clar. També es pot observar que el temps de GPU no ha canviat; això és degut que el codi executat a la GPU és compilat per un altre compilador, no per `gcc`.

Test de la traducció de la llista d'índexs

Degut a l'estructuració del projecte, alguns dels tests que venen a continuació no es poden realitzar canviant arguments de la línia de comandes, sinó que s'han d'activar o desactivar certes parts del codi manualment.

Començaré analitzant, més breument, l'algorisme de traducció de manera aïllada, utilitzant 1 fil, OpenMP i GPU. Per la presència de divergència en aquest algorisme, la codificació amb instruccions vectorials és molt complexa i no l'he realitzada. Els resultats estan plasmats a la taula 4.5. Els paràmetres d'entrada són els mateixos que en el test anterior.

Mètode	Temps (ms) (-g)	Temps (ms) (-O3)
1 fil	669.11	235.69
OpenMP	129.91	45.65
GPU	**20.66**	**20.66**

Taula 4.5: Comparació dels diferents mètodes de traducció de la llista d'índexs. La segona columna presenta els resultats amb el programa compilat sense optimitzacions i la tercera, amb optimitzacions.

Seguidament, crec que és interessant comprovar què passa quan es mesclen els dos algorismes en 1 de sol; com que els dos recorren la mateixa llista, és molt fàcil agrupar-los. A la taula 4.6 exposo els resultats compilant el projecte sense optimitzacions i a la taula 4.7, amb optimitzacions.

Mètode	Creació	Traducció	Suma	Algorisme combinat
1 fil	70.58	669.11	739.69	731.50
OpenMP	25.31	129.91	155.22	150.51
GPU	**4.18**	**20.66**	**24.84**	**20.66**

Taula 4.6: Taula on es mostren els resultats d'executar els algorismes de creació o traducció de la llista d'índexs de diferents maneres sense optimitzacions del compilador. També es mostra el temps dels dos algorismes seqüencials i el temps combinant programàticament els algorismes. Les unitats són mil·lisegons (ms).

Mètode	Creació	Traducció	Suma	Algorisme combinat
1 fil	16.82	235.96	252.78	240.13
OpenMP	21.91	45.65	67.56	62.43
GPU	**4.18**	**20.66**	**24.84**	**20.66**

Taula 4.7: Taula on es mostren els resultats d'executar els algorismes de creació o traducció de la llista d'índexs de diferents maneres amb optimitzacions del compilador. També es mostra el temps dels dos algorismes seqüencials i el temps combinant programàticament els algorismes. Les unitats són mil·lisegons (ms).

Com es pot apreciar a les taules, el fet de combinar algorismes acaba donant més bons resultats, ja que només es llegeixen un cop les dades de memòria.

Test de creació de primers

Seguidament, vull comentar per sobre l'algorisme que utilitza la llista d'índexs traduïda i el sedàs original per trobar tots els primers compresos en un rang concret de valors.

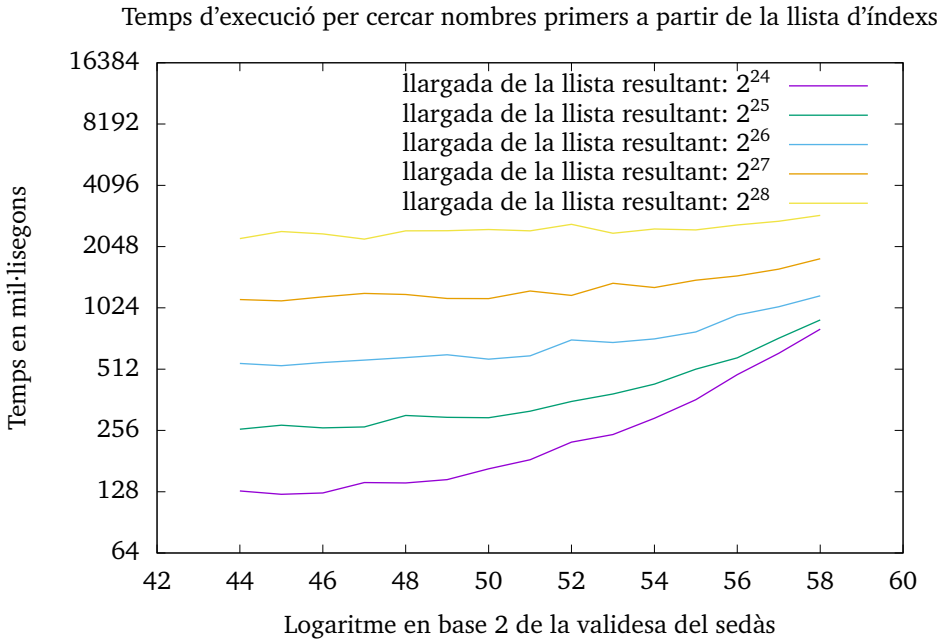

Figura 4.8: Relació del temps emprat segons: la mida del rang de primers on es cercarà i la mida del sedàs. Per trobar primers menors que n, cal trobar els primers menors que $\sqrt{n}$. L'eix horitzontal representa el logaritme en base 2 d'n.

A la figura 4.8 es pot observar les diferències en temps de càlcul variant els paràmetres d'entrada. El test busca els nombres primers que hi ha a partir de 2^{45} en un rang que he fet variar entre 2^{24} i 2^{28}. Tal com he explicat a l'apartat 4.2.3, per executar l'algorisme es necessita una llista prèvia de primers, que va des de 3 fins a l'arrel quadrada del nombre més gran que volem calcular, n. He fet variar n i l'eix x representa el $\log_2 n$.

Com es pot observar, duplicant la mida del rang, es duplica el temps de càlcul, per tant, la relació entre el rang i el temps de càlcul és lineal. A més a més, l'increment de la mida del sedàs implica també un increment en el temps de càlcul i la relació és exponencial.

4.2.5.2 Tests globals

L'objectiu d'aquest seguit de tests és avaluar el mode d'execució de *collaborative computing* comparant-lo amb el mode iteratiu. Les dades d'entrada coincideixen amb les dels dos primers tests. L'únic que afegiré és l'argument `--iterations=10` per executar 10 iteracions. El projecte estarà generat en mode *Release*, és a dir, amb les optimitzacions del compilador. Les proves que realitzaré seran les següents:

1 fil Executaré tots els algorismes (creació i traducció de llista d'índexs i generació dels primers) utilitzant un sol fil d'execució, en el mode iteratiu.

OpenMP Executaré tots els algorismes (creació i traducció de llista d'índexs i generació dels primers) utilitzant tots els fils disponibles amb OpenMP, en el mode iteratiu.

GPU Executaré els algorismes de creació i traducció de llista d'índexs a la GPU i la generació dels primers, a la CPU, utilitzant tots els fils disponibles amb OpenMP. Utilitzaré també el mode iteratiu.

Collaborative computing Executaré el programa en mode computació col·laborativa, utilitzant 8 consumidors amb una cua d'11 posicions. La GPU executarà les tasques de creació i traducció de la llista d'índexs. Cadascun consumidors de CPU executarà l'algorisme de generació dels primers utilitzant només un fil, ja que el processador només disposa de 8 nuclis virtuals.

***Collaborative computing* amb OpenMP** Es tracta del mateix que l'anterior test, però cada consumidor utilitzarà 8 fils per la generació dels primers.

A la taula 4.8 es pot comprovar que l'execució amb el mode *collaborative computing* és la que dóna millors resultats, al paral·lelitzar certes parts del càlcul amb altres.

Mètode	Temps mitjà per iteració (segons)
1 fil	1.79
OpenMP	1.69
GPU	1.74
Collaborative computing	**1.33**
Collaborative computing amb OpenMP	1.38

Taula 4.8: Comparativa entre els diferents mètodes esmentats anteriorment. El temps mitjà per iteració està calculat dividint el temps emprat per tot el càlcul (exceptuant el còmput de la llista de primers que es realitza al començament de l'execució) entre el nombre d'iteracions, 10, en aquest cas.

També és una bona pràctica inspeccionar les execucions amb un *profiler*. Com que es tracta d'OpenCL, NVIDIA no proporciona cap mena de suport per aquests programes a les seves targetes gràfiques. La figura 4.9 és una captura d'una execució en mode *collaborative computing* amb uns paràmetres semblants en una altra màquina. Es pot observar a la línia de temps les diferents crides al *driver* d'OpenCL i les diferents execucions dels *kernels*.

Com es pot veure a la figura 4.9, existeix una desproporció considerable entre les tasques de CPU i GPU. Degut a la falta de recursos i a la desproporció de la tasca de CPU i GPU, el solapament de tasques no és gaire evident; però tot i així, s'aconsegueix solapament a nivell de CPU.

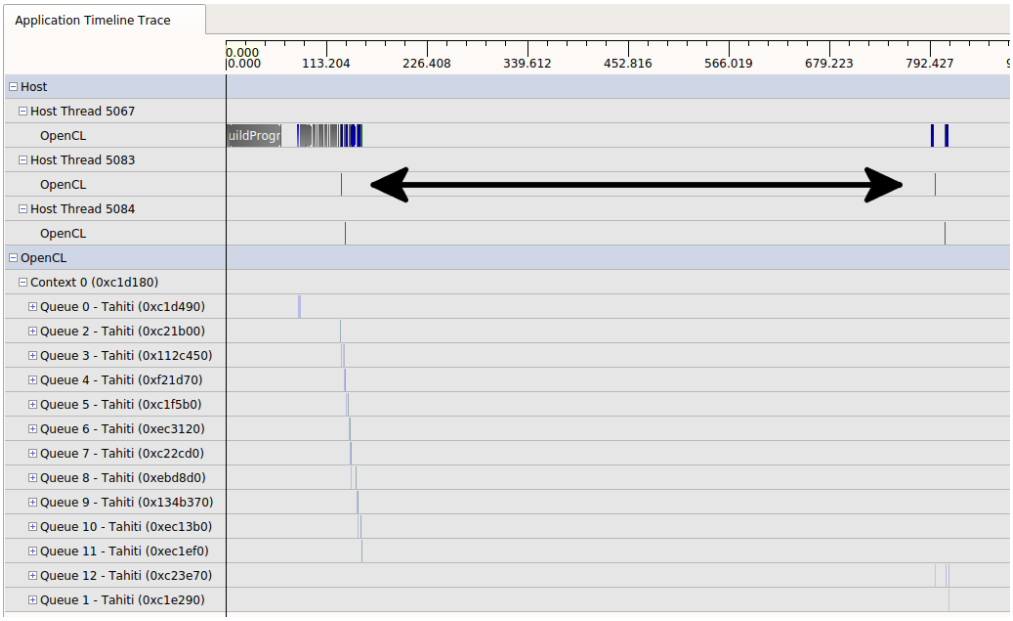

Figura 4.9: Execució amb uns paràmetres semblants del mateix codi executat en aquest test. La secció superior representa l'hoste, amb les crides al *driver* d'OpenCL i la secció inferior és l'anàlisi de l'execució de les crides a la targeta gràfica. L'espai marcat amb la doble fletxa representa el temps d'execució de la tasca en GPU, i les petites ratlles de color blau de la secció inferior representen les execucions dels *kernels*.

Com que NVIDIA no ofereix suport per analitzar aplicacions que utilitzen OpenCL, la captura de la figura 4.9 ha sigut feta utilitzant un processador Pentium 4 de 2 nuclis sense *Hiper-Threading*, 4 GB de memòria RAM i la targeta gràfica Radeon HD7970, en el sistema operatiu Ubuntu 14.04.2.

4.2.6 Conclusions dels resultats

Començaré comentant els resultats obtinguts per extreure'n conclusions, i d'aquí, possible feina a realitzar per optimitzar més el codi.

4.2.6.1 Conclusions dels tests individuals

Com es pot observar a la taula 4.3, la utilització de 8 fils per solucionar el problema no resulta en un vuitè del temps emprat usant un fil. Hi ha diferents motius pels quals pot succeir:

- El processador té 4 nuclis; la tecnologia *Hyper-Threading* no multiplica el nombre de fils, sinó que s'ocupa que el *pipeline* d'execució de cadascun dels nuclis estigui sempre

ple. Tal com internament està connectat, aquest sistema equival a tenir dos fils per nucli.

- Els nuclis, tot i ser independents, comparteixen recursos, com la memòria cau de nivell 3 i els busos de connexió amb la memòria RAM. El fet de que un nucli invalidi una línia de memòria cau al nivell 3, afecta a la resta de nuclis, que hauran d'esperar-se a que la informació vàlida flueixi pels diferents nivells de memòria cau, o fins i tot, a que sigui llegida de memòria RAM.
- Finalment, durant els tests, el meu programa no era l'únic procés executant-se a la màquina, si no que hi havia el sistema operatiu, navegador d'Internet, l'editor de codi, uns quants terminals oberts, etc. Cadascun d'aquests processos, tot i funcionar en segon pla, requereixen d'un petit temps de CPU, que el sistema operatiu els ha de proporcionar, desviant alguns recursos del meu programa.

També vull comentar un fet curiós que es pot veure a la taula 4.4: L'execució emprant 1 fil és més ràpida que emprant 8 fils. Aquesta execució tenia activada l'opció d'optimització `-O3`. Si es fa un cop d'ull a *Options That Control Optimization* [11], l'opció `-O3` activa també `-ftree-loop-vectorize`, on el compilador intentarà vectoritzar certs bucles senzills. Doncs en aquest cas, el compilador ha utilitzat instruccions SIMD per optimitzar el bucle `for`, de la mateixa manera que ho he fet jo explícitament en l'altre versió de l'algorisme. Com que la versió paral·lela està tractada d'una altra manera per OpenMP, és possible que aquest no hagi realitzat cap optimització significativa i per això resulta ser més ineficient.

Aprofitant el moment, vull comentar la diferència entre la programació paral·lela i la programació vectorial. Com s'ha pogut observar, la programació vectorial dóna molt bons resultats i, el més impressionant de tot, és que només utilitza un sol nucli del processador. És bo tenir-ho en compte per processadors que disposin de menys nuclis, perquè la programació vectorial donarà més bons resultats que la programació paral·lela amb entorns on el nombre de fils és molt limitat. Ara bé, com ja he comentat, l'ús d'instruccions vectorials depen molt del tipus d'algorisme.

Es pot observar a les taules 4.6 i 4.7 que, generalment, combinar els dos algorismes en un de sol dóna millors resultats que realitzar els dos per separat. Això és degut a que si es realitzen per separat, s'ha d'iterar dos cops la llista; si es fa conjunt, només cal llegir un cop els valors del vector. També, combinant-los, es dóna la oportunitat al compilador de realitzar certes optimitzacions, ja que té un context més ampli del què es vol aconseguir. Tot i així, un no ha d'esperar-se que el compilador optimitzi per ell.

Si hagués de triar una implementació per l'algorisme de crear i traduir la llista d'índexs, em decanto per la versió de GPU, ja que és la més ràpida; però si no fos possible disposar d'aquesta arquitectura, triaria la implementació amb instruccions vectoritzades, ja que són les que aporten menys gestió per part del sistema operatiu. Com a pas extra, es podria implementar la traducció emprant instruccions SIMD, usant màscares per amagar la divergència de codi, d'una manera semblant com es realitza a la targeta gràfica de manera interna.

Pel que fa a l'algorisme de creació de primers a partir de la llista d'índexs i el sedàs original, crec que és el més complicat d'optimitzar, ja que tracta conjunts de dades molt dispersos. He pogut deduir, analitzant els diferents temps d'execució de l'algorisme amb diferents paràmetres d'entrada, que el problema principal és degut a l'enorme quantitat d'excepcions de pàgina a la memòria cau. Aquest algorisme està compost de dos bucles for niats. L'intern, itera la llista final on hi quedaran els primers, però ho fa en salts difícils de predir, i majoritàriament grans (més grans del que hi pot cabre a la memòria cau). A més, a l'estar implementat usant diferents fils d'execució, provoca que cada escriptura que es realitza s'hagi de propagar a cadascun dels nuclis (pel funcionament intern de la memòria cau), empitjorant més el rendiment.

Una possible solució a seguir en un futur és re-implementar aquest algorisme fent que cada fil sigui responsable només d'un tros de la llista final, no d'un tros de la llista d'entrada, que és tal com està implementat actualment. D'aquesta manera, els fils entre sí no s'anul·larien línies de memòria cau i la localitat de dades seria millor.

Implementar aquest algorisme a la GPU seria un desastre, ja que presenta una divergència increïble (no tots els nuclis faran exactament el mateix). En aquest punt puc parlar dels conceptes *scatter* i *gather*. L'estratègia *scatter* agrupa els algorismes on les entrades són si fa no fa consecutives i les sortides, disperses. L'estratègia *gather* és el contrari: les entrades són disperses, però les sortides són consecutives. L'estratègia *gather* és la més eficient a les GPU. Si s'implementa aquest algorisme amb aquesta estratègia a la GPU, la penalització en el rendiment està en la lectura de dades per l'arquitectura de memòria d'aquesta plataforma. Aquesta penalització és molt menor a la CPU. En tot cas, si el nivell de paral·lelisme fos similar, compensat per l'obtenció o l'escriptura de dades, la CPU tindria més possibilitats de ser més ràpida, ja que posseeix predicció de salt i funciona a més altes freqüències.

4.2.6.2 Conclusions dels tests globals

Els diferents tests realitzats han servit per comparar l'execució que fa ús de *collaborative computing* amb l'execució seqüencial més tradicional. Esperava que el mode iteratiu fos més lent, ja que no hi ha cap paral·lelisme explícit entre algorismes. Com es pot observar a la taula 4.8, l'execució usant un sol fil és la més lenta; utilitzant OpenMP en els algorismes que ho permeten, millora el rendiment, però no el que un s'esperaria. Aquest fet és degut que cada algorisme crea i destrueix els seus fils; crear i destruir fils és una tasca costosa a nivell de sistema operatiu. Una possible solució seria implementar-ho de tal forma que els fils només es creessin una vegada i s'anessin utilitzant (*thread pool*).

Es pot observar també que quan s'utilitza la targeta gràfica, que és el mètode que presentava millors resultats, no és precisament el més ràpid, ja que la sincronització entre hoste i GPU provoca temps d'espera a la CPU que podria estar utilitzant per fer altres tasques. I, és aquí, on entra la computació col·laborativa: mantenir la gestió entre hoste i GPU asíncrona de manera que cap dels dos dispositius espera l'altre.

Es pot veure que els tests amb computació col·laborativa són els que donen millors resultats. El primer test, cada consumidor realitzava les seves tasques emprant un sol fil, i en el segon, cada consumidor les realitzava emprant 8 fils. Per tant, en el primer cas, hi havia un total de 8 fils actius durant el càlcul i en el segon, 64. Com es pot veure, el segon test, tot i tenir més fils, ha donat pitjors resultats. Això es deu a la feina extra que ha de realitzar el sistema operatiu per gestionar tots els fils; recordo que la màquina només en disposa de 8.

El recurs de la computació col·laborativa s'acostuma a utilitzar en arquitectures on la CPU i la GPU conviuen en el mateix xip, com és el cas de la majoria dels processadors d'Intel i les APU de AMD. Aquest tipus d'arquitectures tenen un sistema de memòria compartida que les fa ideals per aquest tipus de programació.

Als antecedents d'aquest apartat he fet referència a una implementació del sedàs d'Eratòstenes realitzada amb CPU; el creador fa ús de la mida de la memòria cau per crear un vector prou petit perquè hi càpiga sencer. Aquest és un bon exemple de programació conscient de l'arquitectura. També usa altres tècniques algorísmiques per aconseguir més bon rendiment. El que he fet jo és més aviat una exploració, obert a noves propostes i noves tècniques de programació per tal de fer el codi més eficient.

4.3 Programació conscient de l'arquitectura a l'empresa

En aquest apartat, exploraré l'aplicació de la programació conscient de l'arquitectura en el món industrial. La feina feta i exposada en aquest apartat s'emmarca en les pràctiques en empresa que estic fent a Mediapro, al departament de Recerca i Desenvolupament.

L'enfocament d'aquest apartat no serà només tècnic, sinó que també exploratiu. Treballaré amb una arquitectura diferent a la tradicional amb l'objectiu de transformar un projecte existent perquè se'n pugui beneficiar.

Com que es tracta d'analitzar unes tasques realitzades en un entorn privat, per motius de confidencialitat no podré explicar segons quins detalls d'implementacions o característiques internes del producte.

4.3.1 Motivació

L'interès principal i que diferencia aquest apartat de la resta és veure com tots els coneixements adquirits sobre la programació conscient de l'arquitectura es poden aplicar en el món professional.

Dins d'un projecte basat en l'exploració de noves tecnologies per ser aplicades a un producte, he realitzat una recerca de les possibilitats d'una nova plataforma de computació, emmarcada dins del que s'anomena *egde computing*. En aquest marc computacional es realitzen tasques de processament d'imatge i visió per computador en temps real.

L'interès d'aquest projecte per és explorar una nova arquitectura encastada (*embedded*) per fer *edge computing* i que permeti executar un producte mínimament viable podent obrir nous mercats de baix cost.

4.3.2 Antecedents

El departament de Recerca i Desenvolupament de Mediapro ja té un producte que actualment ja utilitza plataformes d'*edge computing* basades en servidors que capturen vídeo i el processen en temps real. Els costs i la complexitat de funcionament i instal·lació fa que els sistemes actuals siguin incompatibles amb certs tipus de mercats.

En general, el processament d'imatges i la visió per computador clàssica són tasques heterogènies; és a dir, que disposen d'una varietat molt àmplia d'algorismes estructuralment molt diferents. Per una banda, tota tasca comença amb el processament de matrius, que és com s'interpreten programàticament les imatges. Els algorismes que se n'encarreguen poden, o bé presentar un alt paral·lelisme de dades o bé ser algorismes de cerca. Els primers poden ser fàcilment executats a la GPU, com he demostrat en certa manera a l'apartat anterior i

els segons, presenten millors resultats essent executats a la CPU. També existeixen algorismes de més alt nivell que treballen amb els resultats que els dos primers han generat, per extreure'n conclusions i prendre certes decisions. Aquests, per la seva naturalesa, solen ser executats a la CPU.

L'equip de desenvolupament ja ha fet una certa exploració del maquinari més adequat per executar l'aplicació; ha centrat els esforços en arquitectures de CPU i de GPU, ja que són les més assequibles i les que presenten millors resultats pel cost de desenvolupament i desplegament. Actualment aquestes dues arquitectures poden coexistir al mateix xip o no.

En sistemes on la GPU està connectada mitjançant PCIe (*Peripheral Component Interconnect Express*), que també s'anomenen sistemes amb GPU discreta, la desplegament de tasques pot arribar a ser molt ineficient degut a la latència de la transferència de les dades entre la memòria de la CPU (memòria RAM) i la memòria de la GPU, ja que com que és un perifèric, no comparteix el mateix espai de memòria. Com a contrapartida, al ser un dispositiu separat, disposa de més espai per aconseguir una capacitat de càlcul superior.

Des de fa temps que han aparegut un nou conjunt d'arquitectures on la CPU i la GPU conviuen fusionades en un mateix xip, podent compartir el mateix espai de memòria i, fins i tot, les memòries cau. L'inconvenient principal és que la GPU disposa d'un ample de banda de memòria molt més reduït que en les GPU discretes i, a més, el bus de dades és compartit amb la CPU. Tot i així, al compartir un mateix espai de memòria, les transferències de dades entre CPU i GPU poden quedar reduïdes al no res, fent que el repartiment de tasques que parlava abans sigui molt més senzill de gestionar i molt més eficient.

L'exploració de noves plataformes s'emmarca dins d'aquest tipus d'arquitectures fusionades, ja que es busca una reducció del cost del maquinari i del cost de desenvolupament, per un programari que ja existeix. Intel, AMD, NVIDIA i Qualcomm són alguns dels fabricants que ofereixen aquest tipus de solucions.

Tornant al projecte a on estic treballant, s'ha de tenir en compte que posseir un codi base gran, implementat en C++ i CUDA (propietari d'NVIDIA), pot implicar un cert cost extra per adaptar-lo a la nova plataforma. Així, s'ha decidit explorar les possibilitats de les arquitectures encastades d'NVIDIA pels següents motius:

- En els casos d'aplicacions que poden fer ús intensiu de la computació col·laborativa, es pot arribar a obtenir, amb un codi bàsic, millor rendiment en una plataforma on la CPU i GPU comparteixen memòria.
- En aquests mateixos casos d'aplicacions, quan es tracta de GPU discrets, cal un treball conscient de l'arquitectura per «amagar» les latències de comunicació entre CPU i GPU.
- Addicionalment, les optimitzacions dels codis de GPU poden canviar entre les petites GPU dels sistemes fusionats i les grans GPU discretes, que disposen d'un ample de banda de memòria molt més ample.

- No existeixen alternatives que tinguin una relació preu/capacitat de càlcul millor que les que NVIDIA ofereix.
- La quantitat de treball requerit per adaptar el projecte a un nou entorn de programació (OpenCL, per exemple) és molt més superior a la que el departament té disponible.

NVIDIA disposa d'una gamma de productes encastats anomenada Jetson. Aquesta plataforma es basa en CPUs d'arquitectura ARMv7 i ARMv8 i GPU d'NVIDIA. A l'inici de les primeres exploracions hi havia disponibles els models TK1 i TX1, però poc després, NVIDIA va presentar el nou model, la Jetson TX2, amb el doble de memòria i capacitat de càlcul, amb moltes altres capacitats millorades. Ara que el prototip de la primera versió del projecte en plataforma encastada està quasi enllestit, NVIDIA torna a anunciar una nova Jetson, anomenada «Xavier», que multiplica per 20 les capacitats de càlcul de la Jetson TX2 gràcies a la nova microarquitectura de GPU *Volta*. Tot i així, en el moment d'escriure aquesta memòria, no està disponible per comprar-la.

La Jetson TX1 posseeix 4GB de memòria RAM, que són insuficients per encabir un mínim de funcionalitats. Molta de la feina inicial haguera sigut l'optimització d'ús de memòria; però al tenir disponible la Jetson TX2 al mateix preu, ràpidament es va decidir adquirir-la i basar el desenvolupament en aquesta nova plataforma.

4.3.3 Migració a la nova plataforma

El producte en sí és un programari molt complex que utilitza conceptes avançats d'orientació a objectes en C++ i components de càlcul altament paral·lel basats en CUDA. A més, té un seguit de mòduls que realitzen tasques molt concretes i especialitzades, com ara l'adquisició de de fotogrames de vídeo de les càmeres o d'arxius o els mòduls de codificació de vídeo o *streaming* per la xarxa. També hi ha altres components dedicats a proporcionar una o diverses interfícies a l'usuari, però en aquesta exploració, només analitzaré l'impacte que tenen en el rendiment global, no pas les seves funcionalitats o tecnologies emprades.

Aquest serà el programari que haurà de poder-se executar a la plataforma Jetson TX2. El primer pas, caldrà fer funcionar només una part d'aquests mòduls en aquesta plataforma, però s'hauran de tenir en compte diferents factors molt determinants.

4.3.3.1 Sistema operatiu

El sistema operatiu és el macro-programa que interacciona amb el maquinari i proveeix de serveis a les aplicacions d'usuari. Gestiona els recursos de la màquina perquè diferents programes puguin funcionar a la vegada, proporcionant una interfície senzilla per a les funcionalitats de la CPU i també determina el comportament de molts altres components de programari, com per exemple, els controladors de dispositius (*drivers*) que permeten

utilitzar la GPU. Per tant, el sistema operatiu té una gran influència sobre la programació conscient de l'arquitectura.

El sistema operatiu que s'utilitza actualment per l'aplicació és Windows 10, però la plataforma Jetson TX2 només funciona en Linux (Ubuntu 16). En part, és bo que utilitzi Linux, ja que en el software original hi ha molta feina per solucionar els problemes de rendiment que genera el WDDM (*Windows Driver Display Manager*).

L'ús de Linux em farà tenir en compte que les llibreries de terces que utilitza el programa estiguin disponibles per aquest sistema operatiu. Els repositoris d'Ubuntu són força grans, però és molt habitual que el programa requereixi una versió específica de la llibreria. Per aquest motiu, la majoria de les llibreries les he hagut de compilar a partir del codi font.

Per altra banda, hauré de revisar el codi font per desactivar o substituir aquelles parts que només funcionen en Windows (ús de la API de Windows). Quan es va iniciar aquest projecte, ja es va pensar que tard o d'hora s'acabaria utilitzant Linux com a sistema operatiu, i per això, només algunes parts de codi molt específiques (connectivitat amb *sockets* per la xarxa, per exemple) han portat problemes de compatibilitat.

Gran part de la meva estada al departament ha estat dedicada a fer que el producte es compili i s'executi en Linux, i més concretament, a la plataforma Jetson TX2.

4.3.3.2 Arquitectura del sistema de memòria

Com he comentat anteriorment, el sistema de memòria de la Jetson està compartit entre la CPU i la GPU i altres components (a l'apèndix D.1 hi ha un diagrama on mostra tots els components de la Jetson). Aquest fet implica el següent:

- Cada component disposa de menys ample de banda de memòria, ja que està compartit.
- La memòria que utilitza la GPU és LPDDR4, no pas GDDR5 com la que s'utilitza actualment a les targetes gràfiques discretes. La memòria DDR no està optimitzada pel tipus d'accessos que la targeta gràfica acostuma a fer.
- La proporció en mida de les memòries cau, com a punt positiu, és molt més gran que en una GPU discreta, fent que certs accessos a memòria principal es puguin evitar si la localitat de dades és prou acurada.
- Entre CPU i GPU hi ha molta menys latència, ja que conviuen al mateix bloc de material semiconductor (*die*) o xip. La programació és més simple, ja que s'eviten còpies de memòria, perquè el maquinari permet que la CPU i la GPU comparteixin els mateixos punters.

Al codi original hi ha moltíssimes optimitzacions pensades per solucionar els problemes de rendiment quan es tracta d'una arquitectura amb GPU discreta, però a la plataforma Jetson, algunes d'aquestes optimitzacions no són necessàries i poden empitjorar el rendiment.

Existeixen diferents maneres de gestionar la memòria a la plataforma Jetson, ja que es tracta d'un sistema de memòria compartida:

- **Gestió tradicional**. La CPU i la GPU no comparteixen el mateix espai de memòria (encara que físicament siguin el mateix). Les còpies entre aquests dos dispositius s'han de realitzar explícitament.
- **Memòria *mapped***. Aquest sistema permet compartir els espais de memòria entre CPU i GPU. Els punters a memòria reservats amb aquesta característica activa es podran utilitzar en les dues arquitectures, evitant les còpies explícites de memòria. A la plataforma Jetson, s'ha de desactivar l'ús de memòries cau per utilitzar-ho.
- **Memòria *managed***. Aquest mecanisme de gestió unifica els espais de memòria tant de CPU com de GPU (encara que n'hi hagi més d'una). Es tracta d'un sistema de paginació global de memòria, a un nivell més superior a la memòria *mapped*. També és anomenada *unified memory*. És molt útil en plataformes com la Jetson.

4.3.3.3 Captura de càmera

La placa de desenvolupament de la Jetson TX2 posseeix gran quantitat de connexions per a diferents perifèrics: SATA, PCIe, Ethernet, USB3, MIPI-CSI2, i un llarg etcètera. L'objectiu principal d'aquest projecte, com he mencionat anteriorment, és obrir nous mercats de baix cost, per tant, es necessita una càmera que ofereixi gran qualitat d'imatge en un preu contingut. La connexió de càmera que ho permet és MIPI-CSI2. Però amb la selecció de la interfície de connexió per iniciar el desenvolupament no n'hi ha prou, perquè depen del mòdul de càmera quin serà el format d'imatge i quins seran els controls de captura que posarà a l'abast del programador.

Existeixen un parell de llibreries que simplifiquen molt aquesta captura, i són LibArgus i GStreamer. Cap d'aquestes dues llibreries s'utilitza actualment a l'aplicació, per tant, una de les tasques serà desenvolupar un nou mòdul de càmera per realitzar-ne la captura i tenir un control dels paràmetres principals.

La primera, és una llibreria de baix nivell desenvolupada per NVIDIA per donar al programador una interfície per capturar imatges de la càmera i controlar-ne els paràmetres. La segona, és un *framework* multimèdia obert que permet construir grafs amb elements específics per tractar imatge/vídeo i àudio. NVIDIA ha desenvolupat un mòdul de GStreamer per interactuar amb la càmera connectada per MIPI-CSI2 de la Jetson TX2. Aquesta llibreria és de més alt nivell.

Però, com he dit, no n'hi ha prou amb programar el mòdul de càmera pel producte amb GStreamer; cal conèixer el procés intern. El departament va organitzar una petita conferència amb els desenvolupadors d'NVIDIA per preguntar-los-hi quin era el funcionament intern del maquinari. En resum, la càmera proporciona una imatge en *bayer* (tal com surt del sensor) i un mòdul de maquinari anomenat ISP (*In-System Programming*) que posseeix la Jetson converteix aquesta imatge en el format YUV420.

Per poder treballar amb les imatges, l'aplicació necessita un format d'imatge RGBA. Malauradament, la ISP de la Jetson no ho proporciona i s'ha de fer la conversió programàticament. Explorant amb altres fabricants de càmeres, ens van comentar que el problema de proporcionar el format RGBA és l'ample de banda: una imatge YUV420 ocupa 1.5 canals; una RGBA, n'ocupa 4. Ens van proposar maquinari que proporciona imatges RGB, però encara no ha arribat.

4.3.3.4 Codificació de vídeo

Tot i tractar-se d'una plataforma d'NVIDIA, la Jetson TX2 no posseeix el mateix maquinari de codificació de vídeo que les GPU discretes del mateix fabricant, i per tant, la llibreria utilitzada, *nvenc*, no és compatible.

Existeixen dues alternatives: video4Linux2 i GStreamer. Altra vegada, la primera és de baix nivell i GStreamer, no tant. A més, la segona ja l'he utilitzada per realitzar el mòdul de captura de càmera, per tant, par de l'aprenentatge ja està fet.

El problema (o virtut, depen del punt de vista) és que és un sistema tancat; tot el que surti dels estàndards del graf, no se'n garanteix un bon funcionament o un bon rendiment. Tots els fotogrames que es processen en el programa estan en memòria de GPU (encara que sigui el mateix espai). Com he mencionat a l'apartat 4.3.3.2, existeix el sistema de memòria *mapped*, però com explicaré més avall, no sempre és fàcil canviar el comportament de tot el sistema de memòria del programa. Per aquest motiu, he hagut de buscar la manera més eficient de transferir el fotograma de l'espai (virtual) de GPU al de CPU perquè GStreamer el pogués codificar en un *stream* de vídeo.

4.3.3.5 Interfície gràfica

Al tractar-se d'una plataforma on la CPU i la GPU comparteixen el mateix bus de memòria, el fet de tenir un entorn de finestres carregat influeix en el rendiment final de l'aplicació. L'aplicació mostra vídeos per pantalla i s'utilitza OpenGL per fer-ho. Així doncs, s'està consumint part dels recursos de la Jetson per mostrar imatges per pantalla.

El producte final no estarà a l'abast de l'usuari, sinó que estarà en un lloc de difícil accés, ja que la càmera, per la naturalesa de la connexió MIPI-CSI2, ha d'estar juntament amb

la Jetson. Per aquest motiu, no s'interaccionarà ni amb teclat ni pantalla, sinó remotament amb interfícies web o similars, podent eliminar l'entorn de finestres i així, alliberar recursos.

4.3.3.6 Arquitectura de GPU

L'arquitectura de la GPU de la Jetson TX2 està basada en la mateixa microarquitectura que les GPU discretes, la microarquitectura Pascal. Per tant, també disposa del mateix tipus de clústers de computació (SM – *Stream Multiprocessor*), però només en té 2, en contraposició a 30 en una Quadro P4000. A més, té un ample de banda de memòria més limitat.

Per tant, aquesta GPU no es podrà beneficiar d'algorismes que expressin un alt nivell de paral·lelisme de dades, però sí que es podrà beneficiar d'una millor gestió de la localitat de les dades, ja que, en proporció, disposa d'unes memòries cau més grans. Per tant, l'enfocament de les optimitzacions dels *kernels* de GPU canvia al tenir en compte la Jetson TX2. He pogut experimentar aquesta diferència d'arquitectura en el *kernel* de correcció de distorsió i correcció de color.

***Kernel* de correcció de distorsió i de color** Observant *profilings* d'aquest *kernel*, vaig veure que la major part del temps (voi un 75%) la GPU estava esperant dades, en forma de textures, emmagatzemades a la memòria principal. L'accés aleatori no era suavitzat per les memòries cau de textura.

Les memòries cau de textura, a diferència de les normals, estan organitzades en dues dimensions, de manera que es parla de localitat de dades en dues dimensions. Gràcies a aquest tipus de maquinari, l'accés a textures mitjançant coordenades cartesianes està molt optimitzat i fa que els *thread blocks* quadrats funcionin millor que els rectangulars.

La solució que vaig aplicar per aconseguir més localitat de dades en els accessos per aprofitar les memòries cau de textura va ser crear menys *thread blocks* que realitzessin més feina localment. Per exemple, vaig crear *thread blocks* de 16×16 que treballen 4 porcions, també de 16×16, consecutives en l'eix horitzontal o vertical de la imatge. Així, la localitat de dades augmenta, fent ús de les memòries cau de textura i disminuint la latència d'accés a memòria global. Degut a que els *thread blocks* es distribueixen de forma no controlada pel programador, es pot donar el cas que *thread blocks* que treballin sobre dades properes s'executin en *Stream Multiprocessors* diferents o el revés, empitjorant l'ús de les memòries cau de textura. Seguint l'estratègia comentada, la localitat de dades queda assegurada.

Existeix una segona optimització, però no a nivell d'algorisme i accés a les dades sinó a nivell funcional. La interfície de connexió de càmera de la Jetson disposa d'una ISP que realitza tasques de correcció de color. Les proves realitzades amb programari de tercers per avaluar la qualitat d'imatge de la càmera revelaven que tant la qualitat d'imatge com la fidelitat dels colors era perfecte per l'ús que se li volia donar. Per aquest motiu, no és necessari realitzar

una correcció del color, ja que la que proporciona el maquinari és acceptable. Per això, aquesta segona optimització elimina aquesta part del *kernel*.

I fins aquí el resum de la meva tasca al departament de Recerca i Desenvolupament de Mediapro. Seguidament, explicaré alguns dels resultats que he obtingut al fer el canvi de sistema operatiu, al fer servir noves llibreries pel tractament multimèdia i les millores aconseguides en l'optimització d'un *kernel*.

4.3.4 Proves i resultats

En aquest apartat exposaré les proves i resultats que he anat realitzant al llarg de l'estada al departament. Per mantenir un ordre concret, ho organitzaré per seccions, tal com ho he fet a l'apartat anterior.

4.3.4.1 Sistema operatiu

Les millores de rendiment utilitzant Linux han sigut més notables a l'entorn de desenvolupament que a l'aplicació en sí, però crec que és convenient esmentar-les:

- Els temps de compilació han millorat moltíssim. L'equip de desenvolupament utilitza un programari per repartir tasques de compilació entre tots els ordinadors del departament, disposant en total d'uns 60 nuclis per compilar. El temps emprat per una compilació completa és d'uns 10 minuts. Els temps emprat compilant només utilitzant la Jetson (6 nuclis), és d'uns 45 minuts. Els temps de compilació s'han reduït a la meitat, en aquesta plataforma (tenint en compte el nombre de nuclis). També, si ho comparo amb els resultats en una torre de 8 nuclis executant Linux, el temps total és d'11 minuts, unes 6 vegades menys tenint en compte el nombre de nuclis.
- Les diferències amb les *drivers* són notables: a la figura 4.10, es mostra la línia de temps corresponent al final del processament d'un fotograma. Es pot observar que l'espai que existeix entre la primera crida a la API per executar el *kernel* (de color groc clar, a la fila *Runtime API*) i l'execució del mateix *kernel* (també de color groc clar, a la fila *Compute*) és significativa. Això es deu a que el WDDM no es comunica amb la GPU fins que no té un seguit de crides acumulades.

  A la figura 4.11, es pot observar que la relació entre la crida a la API i l'execució del *kernel* és quasi immediata.

  El nivell de zoom de les dues imatges és equivalent. També es pot observar el nivell de solapament entre les dues plataformes. A la plataforma Windows, com que el maquinari és molt més potent, té la capacitat de solapar l'execució dels *kernels*; a la plataforma Jetson, com que la capacitat de càlcul és més limitada, l'execució dels *kernels* és més serialitzada.

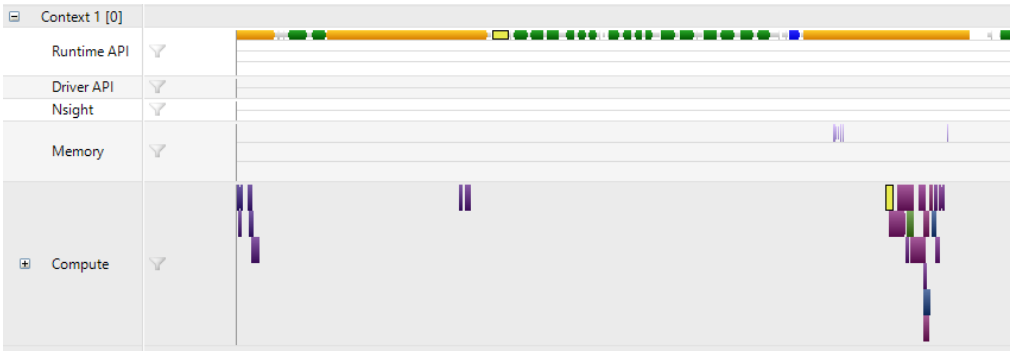

Figura 4.10: *Profiling* de l'aplicació utilitzant Windows. Aquesta imatge pertany a la línia de temps corresponent al final del processament d'un fotograma. Les marques de color verd a la primera fila denoten crides per enfilar *kernels*. Les marques de color morat a la fila de *Compute* denoten l'execució dels *kernels*.

4.3.4.2 Arquitectura del sistema de memòria

A l'apartat 4.3.3.2, he parlat del sistema de memòria *mapped*. El punt clau d'aquest sistema és evitar còpies explícites de memòria entre CPU i GPU, però amb el desavantatge que les memòries cau queden desactivades. Vaig activar aquest tipus de memòria a l'aplicació i vaig canviar la manera que es reservava la memòria perquè no es realitzessin còpies innecessàries. Però els resultats no van ser satisfactoris, ja que al no poder emprar memòries cau va causar una davallada de rendiment.

4.3.4.3 Captura de càmera

Un cop realitzat el mòdul de càmera, aquest va funcionar perfectament sense cap signe que indiqués necessitat d'optimització. GStreamer, internament, ja és eficient i la interfície que mostra al programador ja està optimitzada. Tampoc té molt sentit comparar aquest mòdul amb d'altres de similars, ja que tant la càmera com la connexió son completament diferents.

4.3.4.4 Codificació de vídeo

Tot el processament dels fotogrames es realitza íntegrament a la GPU: quan un fotograma nou arriba, es copia a l'espai de memòria de la GPU. Com he mencionat a l'apartat 4.3.4.2, l'ús de memòria *mapped* per tota l'aplicació no ha sigut possible a no sé que es reescrigui tota, cosa que no és possible. Per tant, de moment s'hauran de mantenir les còpies.

La capa que GStreamer exposa al programador resideix a l'espai de CPU, i per aquest motiu, no hi ha manera senzilla de proporcionar fotogrames per la codificació que resideixin en un espai de memòria diferent (encara que físicament siguin el mateix). Per això, és necessària una còpia de l'espai de GPU al de CPU abans que GStreamer pugui utilitzar el fotograma.

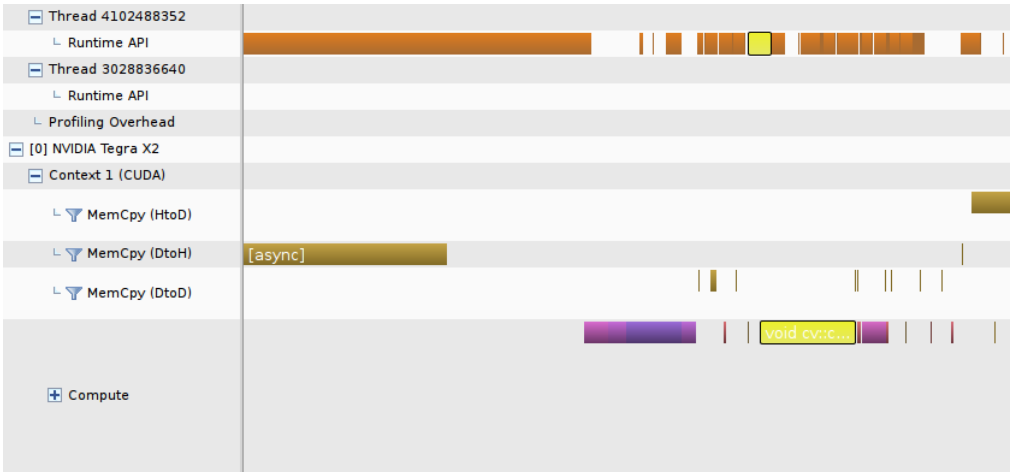

Figura 4.11: *Profiling* de l'aplicació utilitzant Linux a la plataforma Jetson. Aquesta imatge pertany a la línia de temps corresponent al final del processament d'un fotograma. Les marques de color taronja representen les crides per enfilar *kernels* i les marques de color morat, l'execució dels mateixos.

Per minimitzar els temps de sincronització i de còpia, aquesta es realitza just abans de codificar un fotograma: de la GPU a l'espai que GStreamer ha reservat per tal fotograma. D'aquesta manera, es permet que l'aplicació pugui re-usar l'espai de memòria que ocupava i també es cedeix la propietat de la còpia a GStreamer, perquè l'alliberi quan més li convingui.

4.3.4.5 Interfície gràfica

Mentre solucionava alguns problemes de rendiment general, no vaig parar-me a pensar en la interfície gràfica ni en l'entorn gràfic del sistema operatiu, ja que és necessari per configurar l'aplicació. Però tenint en compte que el bus de dades utilitzat per la pantalla està compartit amb el de la GPU i altres components essencials, com el codificador de vídeo i que s'utilitza OpenGL per mostrar les imatges a la interfície, fa pensar que l'ús d'aquesta modalitat d'execució pot perjudicar el rendiment.

L'ús de *profilings* i monitorització dels recursos del sistema revelava que els busos de dades estaven sobre-saturats i que les còpies de memòria ocupaven més temps del necessari. Reduir la qualitat d'anàlisi no va ajudar a millorar el rendiment.

Per tant, el següent pas és executar l'aplicació per consola un cop configurada. El rendiment va millorar significativament. En part és per l'absència d'OpenGL, que acostuma a ser incompatible amb CUDA[6]. Gràcies a aquesta petita optimització, es pot activar la codificació de vídeo juntament amb un anàlisi moderat sense perdre cap fotograma.

[6]Algunes targetes gràfiques d'NVIDIA posseeixen dos modes de funcionament: utilitzant OpenGL o utilitzant CUDA. Per això, quan s'utilitzen les dues llibreries en el mateix programa, la targeta gràfica ha de realitzar una espècie de canvi de context, que succeeix a cada fotograma, sacrificant el bon rendiment de l'aplicació.

4.3.4.6 Arquitectura de GPU

Finalment, cal parlar d'un dels *kernels* que tenen un gran impacte en el rendiment final de l'aplicació.

***Kernel* de correcció de distorsió i de color** Després de la optimització esmentada a l'apartat 4.3.3.6, vaig aconseguir eliminar part de les dependències de memòria. Es pot comprovar observant el diagrama de latències de la figura 4.12 amb el de la figura 4.13.

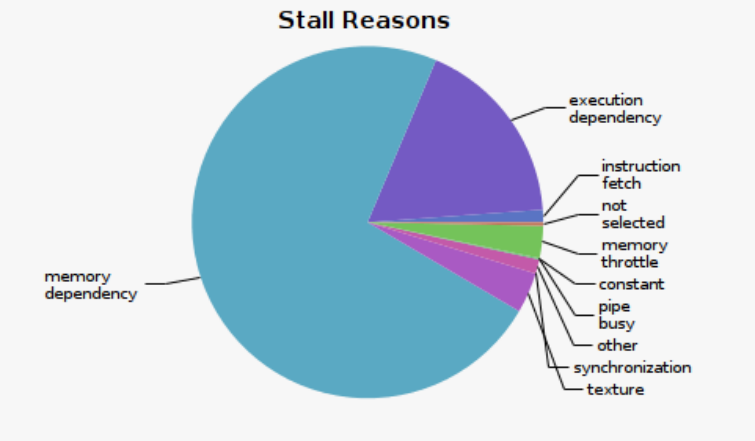

Figura 4.12: Diagrama de latències del *kernel* de correcció de distorsió i de color extret del *profiling* sense cap optimització.

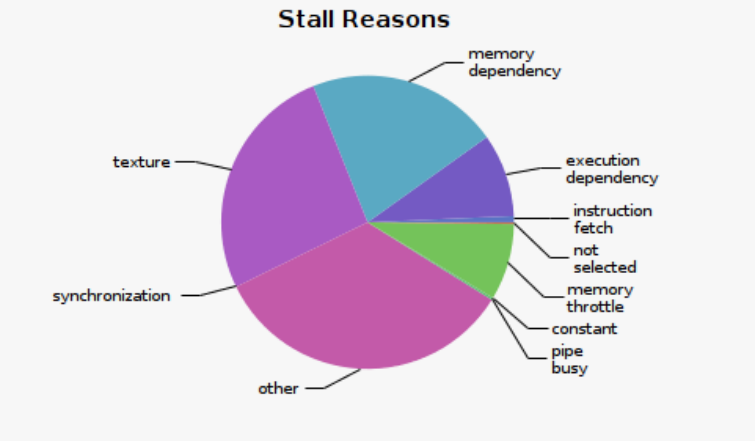

Figura 4.13: Diagrama de latències del *kernel* de correcció de distorsió i de color extret del *profiling* un cop aplicada la optimització esmentada.

A la figura 4.13 es veu que part important de la latència és degut a l'accés a les memòries cau de textura. Reduir aquest temps és inevitable, ja que s'hi ha d'accedir sí o sí, però he reduït moltíssim el temps d'espera a memòria, que és el que realment causava ineficiències. Però existeix una categoria amb un percentatge notable, «altres», que segons Smith [13], l'ocupen aquest seguit de problemes:

- Conflictes en el banc de registres per l'ús de moltes variables vectorials.
- Espera dels *warps* en codis amb branques.
- L'espera de *warps* de prioritat més baixa per a ser executats.

Com que al *kernel* s'utilitzen força variables vectorials i no hi ha branques (creades amb `ifs`, per exemple), és molt segur que la major part d'espera sigui deguda a conflictes en el banc de registres. Però per l'algorisme, no són fàcilment solucionables.

La resta de porcions del diagrama, que ara són més grans, és degut a la proporció. La dependència de memòria ha disminuït moltíssim fent que altres problemes semblin més evidents.

Abans de realitzar la segona optimització, l'ús de textures per a la correcció de color era molt alt. Tal com s'implementa aquesta correcció, els accessos són pràcticament aleatoris, fent que sigui complicat beneficiar-se de les memòries cau de textura. Pel que fa als accessos a textures per corregir la distorsió de la imatge, aquests són molt més locals.

Quan vaig aplicar la segona optimització, els accessos a les textures de color van ser eliminats, reduint moltíssim les dependències de memòria. Es mostren els mateixos diagrames de latències per l'algorisme no optimitzat sense la correcció de color a la figura 4.14 i amb l'algorisme optimitzat, també sense la correcció de color a la figura 4.15.

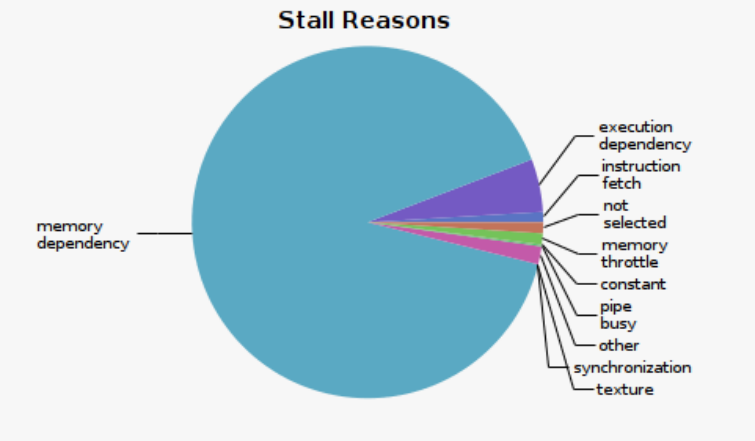

Figura 4.14: Desglossament de les latències del *kernel* de correcció de distorsió i color original (no optimitzat), amb la part de correcció de color desactivada.

Un cop eliminada la part de correcció de color, que implicava certs càlculs matemàtics, roman la latència deguda a l'espera de les dades corresponents a la textura per desfer la distorsió de la imatge. Quan es combinen càlculs matemàtics amb accessos a memòria, la latència queda «amagada». Si la major part del temps els *warps* estan calculant, deixen el bus d'accés a memòria global disponible perquè altres *warps* puguin accedir més ràpidament a la memòria.

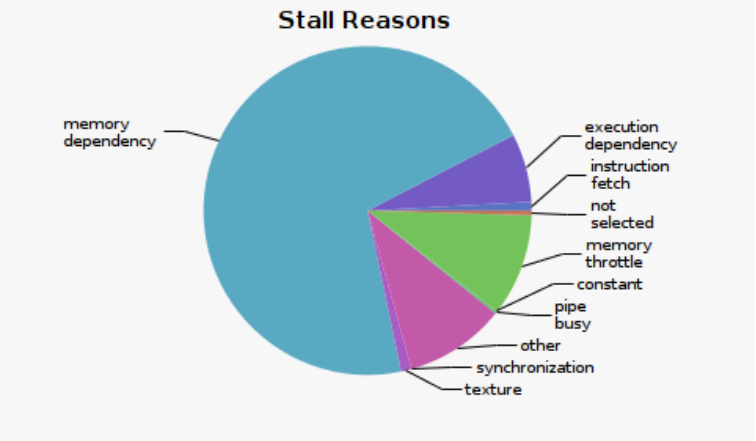

Figura 4.15: Desglossament de les latències del *kernel* de correcció de distorsió i color optimitzat, amb la part de correcció de color desactivada.

Observant les figures 4.14 i 4.15, la dependència de dades sembla haver crescut moltíssim respecte a les anàlogues 4.12 i 4.13. Aquest fet es deu a que l'algorisme de correcció de distorsió no implica càlculs matemàtics, i per tant, no hi ha manera d'amagar la latència d'accés a la memòria. Comparant la versió no optimitzada i la optimitzada, la dependència de dades disminueix gràcies a l'increment de la localitat de dades, poden aprofitar millor les memòries cau de textura. Per acabar aquesta subsecció, presento una taula resum amb els resultats aconseguits:

Mètode	Temps (ms)
Algorisme original	10.7
Algorisme original optimitzat	10.1
Algorisme original sense color	4.58
Algorisme original optimitzat sense color	3.51

Taula 4.9: Aglutinació dels diferents resultats obtinguts amb les dues optimitzacions realitzades. Quan em refereixo al terme «sense color», vull anomenar el *kernel* que té la part de correcció de color desactivada.

Com es pot observar a la taula 4.9, la diferència de temps aconseguida amb la primera optimització no és gaire significativa, però l'aconseguida amb la segona, ho és molt. Reduint el temps i els recursos emprats per aquest *kernel*, es pot aconseguir un rendiment global major o fins i tot, aconseguir realitzar un anàlisi més complex.

4.3.5 Conclusions dels resultats

Com he pogut mostrar a l'apartat de resultats (4.3.4), és important conèixer el maquinari per aconseguir un bon rendiment de l'aplicació. Esmentaré quatre casos, on la programació conscient de l'arquitectura ha tingut un impacte notable en el rendiment de l'aplicació. Aquests són: el sistema operatiu, l'arquitectura de memòria, el maquinari específic i l'arquitectura de la GPU.

Sistema operatiu He pogut mostrar que depenent del *driver* que s'utilitzi i en quin sistema operatiu s'executi, el rendiment pot variar. Crec que és important notar que Linux és un sistema operatiu altament modularitzable, contràriament a Windows. Gràcies a aquest fet, es podria crear una versió pròpia de Linux amb els paquets mínims perquè l'aplicació pugui funcionar. D'aquesta manera, es redueix la càrrega general del sistema ja que hi ha menys tasques en segon pla; també es redueix significativament l'ús de memòria RAM i s'aconsegueix un entorn menys carregat que pot proporcionar més recursos per l'aplicació.

Arquitectura de memòria Els resultats obtinguts utilitzant el sistema de memòria *mapped* no han sigut satisfactoris. Com a treball futur, s'hauria de realitzar un *refactoring* complet de l'aplicació per implementar el sistema de memòria *managed* des de zero, ja que permet mantenir activades les memòries cau i tenir un control més precís de la gestió per a memòries compartides.

Maquinari específic Conèixer maquinari específic integrat al sistema pot proporcionar grans avantatges. Fent referència a l'apartat 4.2, coneixent les instruccions vectorials permet aconseguir un increment molt notable en el rendiment. En el cas de la Jetson, conèixer l'existència i el bons resultats de la ISP, ha permès fer que el *kernel* més costós de tota l'aplicació quedi reduït a molt més de la meitat; d'aquesta manera, es pot aconseguir incrementar la complexitat de l'anàlisi sense sacrificar-ne la qualitat.

Arquitectura de GPU En quan a l'estratègia de *collaborative computing*, els resultats de l'apartat 4.2 han sigut bons, però en aquesta aplicació, són molt més evidents: el solapament de tasques de CPU i GPU és real i, fins i tot, a la GPU es poden arribar a solapar alguns *kernels*. És un cas habitual que la CPU estigui processant els resultats de l'anàlisi del fotograma anterior executat a la GPU. En el cas dels servidors de producció, que suporten fins a 8 càmeres, la càrrega i processament dels fotogrames està solapada; mentre el fotograma de la vuitena càmera s'està copiant a l'espai de memòria de GPU, el fotograma de la primera ja s'ha acabat d'analitzar, i, segurament, la CPU ja està començant a rebre els següents fotogrames. En aquest tipus d'aplicacions, *collaborative computing* és fa molt més evident, i fins i tot, necessari.

També és molt necessària la utilització de *profilers*, per esbrinar quins són realment els colls d'ampolla de l'aplicació. Permeten generar línies de temps com les de la figura 4.10, podent visualitzar quins *kernels* o quines transferències de memòria ocupen més temps o quines parts no queden solapades per algun motiu. Però també permeten generar informes molt concrets sobre l'ús que els *kernels* fan del maquinari com és el diagrama de la figura 4.12 i similars. D'aquesta manera, es poden realitzar les optimitzacions pertinents.

Però no sempre les optimitzacions seran beneficioses tant per les GPU discretes com per la Jetson; sense anar més lluny, un dels *kernels* de conversió d'espai de color va ser optimitzat, fa temps, per aprofitar el gran ample de banda agregat que disposen les targetes gràfiques

Quadro (GPU discreta). Però vaig observar que, per l'arquitectura de la Jetson, que no disposa d'un gran ample de banda de memòria, el codi no optimitzat funcionava millor. Això es deu a que les còpies explícites extra dins el *kernel* tarden molt més a la Jetson ja que no pot aprofitar l'ample de banda agregat que exposa l'optimització.

Finalment, vull mencionar la diferència que hi ha entre executar els *kernels* de manera aïllada i conjuntament amb la resta de l'aplicació. Les millores que es poden observar a la taula resum 4.9 no semblen gaire significatives; però si es prenen mesures del temps d'execució amb la resta de l'aplicació, es pot observar que la versió no optimitzada tarda voi un 25% més. Quan s'executa l'aplicació, els busos de dades mostren un percentatge d'ús molt elevat; si els *kernels* fan un ús més moderat de la memòria aprofitant-se de les memòries cau (localitat de dades), podran funcionar més ràpidament.

Capítol 5

Conclusions

Bé, per concloure aquest treball, m'agradaria esmentar què he aconseguit realitzant-lo i també fer referència a algun comentari a nivell personal que canviaria si tornés a realitzar aquest treball o l'hagués d'ampliar.

Primer de tot, dic el que és obvi: he ampliat els meus coneixements tècnics sobre el que fa referència a l'arquitectura de maquinari. He conegut noves arquitectures, he realitzat programes senzills per ser-hi executats, he conegut i aplicat un estil de programació conscient de l'arquitectura i finalment, he viscut l'aplicació professional de la programació conscient de l'arquitectura.

A la introducció he mencionat que m'agradava el paral·lelisme, i, tot i de vegades no pensar que estigui lligat a l'arquitectura, ho està molt més del que un es pot pensar. També cal dir que el paral·lelisme no mor amb la programació multi-fil o multi-procés; existeixen altres maneres de paral·lelitzar codi com més eficaces en certs algorismes.

Pel que fa a la meva estada a l'empresa, ha sigut enriquidora. M'ha ajudat a espavilarme davant de problemes nous, ja que la meva tasca explorativa travessava la frontera del coneixement de l'equip.

També vull mencionar la utilitat dels codis «acadèmics» o senzills prototips; permeten explorar nous conceptes de manera aïllada per després poder-los aplicar al projecte principal. I no m'agradaria mencionar només els codis; el prototip del processador de 8 bits ja ha visitat un institut on hi he fet una xarrada explicant-ne el funcionament a alumnes de batxillerat i ja s'està parlant d'una segona visita.

Finalment, abans d'entrar en l'experiència personal, vull mencionar l'aspecte comunicatiu d'aquesta memòria. M'ha sigut complex i m'ha portat esforç aconseguir una sòlida estructura i organització del material que permetin al lector entendre el que he realitzat durant mesos en unes poques ratlles.

L'experiència professional que ha sigut més colpidora ha sigut el desenvolupament per la plataforma Jetson; quan vaig començar en aquest projecte, l'escassa documentació i el fet d'obrir una nova via d'investigació ha sigut especialment dur, sobretot en els casos en que cap membre de l'equip em podia ajudar per falta de coneixements de la nova plataforma. Tot i semblar una experiència dura, trobo que és extremament enriquidora. M'ha ajudat a espavilar-me, a prendre decisions, a realitzar proves decisives, etc, que duent a terme una altra tasca, potser no haguera experimentat mai.

Una de les preguntes que m'he realitzat a l'acabar el treball és: *Perquè OpenCL a la part del càlcul de nombres primers?* Un dels motius pels quals ho vaig triar és perquè és un projecte *open source*, que pot ser executat en múltiples arquitectures de multitud de fabricants. Però tot i així, el suport a nivell de programari és escàs, i de vegades, no funciona com un s'espera. Per això, la meva reflexió és que haguera hagut de realitzar la part de GPU emprant CUDA, que és propietari d'NVIDIA, però que proporciona un millor suport en termes de programari. A més, pel motiu que sigui, tinc més fàcilment a disposició targetes d'aquest fabricant que no pas de Radeon, que només suporta OpenCL.

I, ja per acabar, la pregunta: *canviaria alguna cosa d'aquest treball?* I la resposta és que sí. Centraria la recerca en un aspecte més concret i hi aprofundiria més.

Agraïments

Vull donar les gràcies a l'equip de desenvolupadors per la confiança que van dipositar en mi per realitzar aquesta indagació en el seu producte i el suport obtingut. Igualment, vull agrair a l'Òscar per haver accedit a dirigir d'una manera tant entusiasta aquest treball, la seva infinita paciència i haver-me donat la tranquil·litat de poder comptar en tot moment amb la seva complicitat.

Bibliografia

[1] *16K (2K × 8) Parallel EEPROMs*. AT28C16. Rev. 0540B–10/98. Atmel. 1998. URL: http://cva.stanford.edu/classes/cs99s/datasheets/at28c16.pdf.

[2] *4-bit Binary Full Adder With Fast Carry*. 74LS238. Rev. 1995. National Semiconductor. Jun. de 1989. URL: http://www.futurlec.com/Datasheet/74ls/74LS283.pdf.

[3] *8-bit addressable latches*. 74LS259. Rev. March 1988. Texas Instruments. Des. de 1983. URL: http://www.ti.com/lit/ds/symlink/sn74ls259b.pdf.

[4] D. J. Bernstein. *primegen*. URL: https://cr.yp.to/primegen.html.

[5] *CMOS Static RAM*. IDT6116xA. Rev. January 2013. Integrated Device Technology. Jul. de 2000. URL: https://www.idt.com/document/dst/6116sala-datasheet.

[6] Ben Eater. *Build an 8-bit Computer from scratch*. URL: https://eater.net/8bit/.

[7] *Inel Intrinsics Guide*. Intel. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#.

[8] Albert Paul Malvino i Jerald A. Brown. *Digial Computer Electronics*. McGraw Hill Education, 1995.

[9] *NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge*. NVIDIA Corporation. URL: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/.

[10] *Octal Bus Transcievers With 3-State Outputs*. 74LS245. Rev. September 2016. Texas Instruments. Oct. de 1976. URL: http://www.ti.com/lit/ds/symlink/sn74ls245.pdf.

[11] *Options That Control Optimization*. Free Software Foundation. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[12] *Segmented Sieve*. Wikipedia. URL: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes#Segmented_sieve.

[13] Greg Smith. *Other Issue Stall Reason displayed by the Nsight profiler*. URL: https://stackoverflow.com/questions/14887807/what-are-other-issue-stall-reasons-displayed-by-the-nsight-profiler.

[14] spel3o. *How to Build an 8-Bit Computer*. URL: http://www.instructables.com/id/How-to-Build-an-8-Bit-Computer/.

[15] Kim Walisch. *Fast C/C++ prime number generator*. URL: https://primesieve.org/.

[16] *xx555 Precision Timers*. xx555. Rev. September 2014. Texas Instruments. Set. de 1973. URL: http://www.ti.com/lit/ds/symlink/ne555.pdf.

[17] Antonio Hernández Zavala et al. «Design of a General Purpose 8-bit RISC Processor for Computer Architecture Learning». A: *Computación y Sistemas* 19.2 (2015), pàg. 371 - 385. DOI: http://dx.doi.org/10.13053/CyS-19-2-1941.

Apèndix A

Informació addicional sobre el processador de 8 bits

A.1 Sistema de rellotge

El sistema de rellotge ha de permetre generar una senyal quadrada de rellotge que freqüència variable i que sigui prou baixa per poder observar els canvis d'estat dels components a ull nu.

Els cristalls de quars que s'usen com a generadors de freqüència van massa ràpid (de l'orde de kHz) i no són tant senzills d'utilitzar com un rellotge basat en un condensador i un comparador. El més conegut és la sèrie *xx555 Precision Timers* [16], més conegut amb el nom *timer 555*. Connectat de certa manera, és capaç de generar una senyal quadrada de freqüència variable. El rang de freqüències és molt ampli, però per un funcionament correcte del processador, aquest temporitzador generarà pols de fins a 20 Hz.

La majoria de polsadors tenen dos contactes metàl·lics que acostumen a rebotar vàries vegades fins a quedar en contacte (que és quan es prem el polsador). Aquest fet fa que generar un únic pols de rellotge sigui impossible. Per això faré servir el mateix circuit integrat com a *debauncer*, és a dir, per eliminar aquestes vibracions i que el pols de sortida sigui únic.

Finalment, caldrà un polsador commutat per seleccionar la senyal automàtica o la manual. Aquest polsador se li haurà d'afegir un altre *timer 555* per evitar els rebots del mecanisme intern. Un multiplexor activat per aquest polsador seleccionarà la senyal manual o automàtica com a senyal de sortida. També disposarà d'un mecanisme per aturar el rellotge programàticament.

A.2 Registres

Els registres estan connectats al bus principal. Per no interferir amb les dades que hi pugui haver, han d'estar-hi físicament desconnectats si no l'utilitzen. Desconnectar-se físicament no és possible, però es pot simular amb un estat d'alta impedància. En lògica binària existeixen dos estats: 0 i 1; sovint, es parla del tercer estat, el d'alta impedància, que significa «desconnectat».

Per mantenir la simplicitat, he utilitzat un *8-bit addressable latches* [3], un circuit integrat que implementa les funcionalitats de registre, però que no disposa del tercer estat; per aquest motiu, he afegit un *Octal Bus Transcievers With 3-State Outputs* [10], que l'implementa i que fa de porta d'entrada i sortida al bus central. Entre aquests dos circuits integrats, hi he connectat leds, que permeten veure en tot moment el contingut intern del registre. Els detalls de les connexions es troben a l'apèndix B.2.1 i B.2.2.

A.3 Sistema de memòria

El sistema de memòria és un dels més complicats de dissenyar. Tot i tenir uns requeriments molt senzills (donada una adreça, recuperar-ne el valor o escriure'n un valor), la necessitat d'incorporar un programador pel processador en dificulta el procediment.

Perquè el processador pugui començar a funcionar, cal que les instruccions i les dades estiguin carregades a la memòria RAM. La memòria RAM és volàtil, per tant, aquesta informació ha d'estar emmagatzemada en algun altre suport, com per exemple, un disc dur, com en els ordinadors actuals. No he implementat aquest tipus de memòria ja que complicaria el disseny; en canvi, he afegit aquest programador, que, mitjançant un Arduino, farà de pont entre un ordinador actual i el processador, per tal de carregar les instruccions i les dades a la memòria RAM abans d'executar el programa.

La memòria RAM utilitza el registre d'adreça de memòria (MAR) per mantenir-ne una còpia estable; pel que fa l'entrada i sortida de dades de la memòria RAM, està internament multiplexada perquè pugui actuar d'entrada o sortida segons el moment. Aquesta entrada/sortida està connectada al bus principal mitjançant un *bus tranciever*. També està connectat a un multiplexor, que permet seleccionar les dades d'entrada o bé del bus central o bé del programador. Així mateix, el registre d'adreça de memòria (MAR) també té l'entrada multiplexada, per poder rebre l'adreça des del bus central o del programador. Per a més informació sobre les connexions, feu un cop d'ull a l'apèndix B.4.

El control d'aquests multiplexors el té el programador mateix; així, quan s'engega, pot prendre control del mòdul de memòria mentre escriu les instruccions i les dades.

A.4 Sistema de sortida

La lògica per controlar un sol dígit de 7 segments a partir d'un codi en binari natural no és extremament complexa, però necessita molts components. Per això, ho he simplificat usant una memòria no volàtil, una memòria EEPROM (*Electrially Eresable Programable Read Only Memory*). Fent servir 8 entrades (nombre en binari) i 7 sortides (7 segments), puc representar tots els nombres des de 0 fins a 255. Però necessito 28 segments, ja que tinc 4 dígits per controlar. Per tant, enlloc d'emprar 4 memòries, una per cada dígit i el signe, multiplexaré la sortida: donat un instant de temps t, només una xifra de les 4 estarà il·luminada. Fent que la multiplexació ocorri ràpidament, serà indistingible a ull nu.

Les memòries EEPROM que utilitzaré són aquestes: *16K (2K × 8) Parallel EEPROMs* [1]. Tenen una capacitat de 2048 paraules de 8 bits o 2 KBytes. Per tant, tinc 11 bits d'adreça i 8 de dades. Utilitzaré els 8 bits menys significatius com l'entrada pel nombre en binari (natural o en Ca2); els dos següents els utilitzarà el multiplexor (2 bits, quatre posicions, que són els 4 dígits que necessito) i l'onzè bit, serà el bit de signe, activable mitjançant una senyal de control, que indicarà si el nombre llegit del bus està o no en Ca2. La sortida, serà la codificació per un dígit de 7 segments. Per clarificar-ho, posaré un petit exemple:

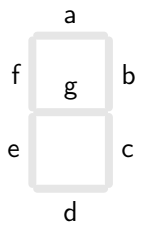

Figura A.1: Nom de cadascun dels leds que formen una xifra.

Xifra	Sortida EEPROM							
	∅	g	f	a	b	c	d	e
	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	1
1	0	0	0	0	1	1	0	0
2	0	1	0	1	1	0	1	1
3	0	1	0	1	1	1	1	0
4	0	1	1	0	1	1	0	0
5	0	1	1	1	0	1	1	0
6	0	1	1	1	0	1	1	1
7	0	0	1	1	1	1	0	0
8	0	1	1	1	1	1	1	1
9	0	1	1	1	1	1	1	0

Taula A.1: Sortida que té l'EEPROM segons el dígit que ha de mostrar. El bit més significatiu correspon al punt decimal que alguna de les pantalles de 7 segmens disposen.

Un cop podem relacionar els bits que s'activaran segons el dígit que s'ha d'il·luminar, tal com mostra la taula A.1 amb l'ajuda de la figura A.1, prendré per entrada un nombre qualsevol, 85 per exemple (01010101 en binari) i mostraré com funciona la multiplexació per aquesta entrada mitjançant la taula A.2:

Com he dit abans, fent que el multiplexor canviï ràpidament de valor, es produeix l'efecte que les 4 xifres estan il·luminades al mateix temps. Per construir aquest multiplexor, he fet servir un *timer 555*, del mateix model que el que s'utilitza al rellotge principal, un comptador

Posició	Adreça										Valor								
	Signe	Multiplexor	Nombre d'entrada (85)								∅	g	f	a	b	c	d	e	
Unitats	0	0	0	0	1	0	1	0	1	0	1	0	1	1	1	0	1	1	0	5
Desenes	0	0	1	0	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	8
Centenes	0	1	0	0	1	0	1	0	1	0	1	0	0	1	1	1	1	1	1	0
Signe	0	1	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	+

Taula A.2: Relació entre els diferents valors del multiplexor i les sortides de la EEPROM per mostrar un valor concret.

de 4 bits (per generar els bits 8 i 9 del multiplexor) i també un demultiplexor, per donar pas de corrent a un dels 4 dígits segons l'entrada proporcionada.

Totes les combinacions que hi ha la memòria EEPROM les he hagut de gravar mitjançant un microcontrolador (Arduino) amb un programa propi que satisfà les característiques mostrades a les taules i figures d'aquest apartat.

A.5 Senyals de control

- Senyals d'escriptura. La seva funció és permetre al mòdul llegir dades del bus central i escriure-les al registre intern. Són les següents:
  - `MAIN`: Memory Address Register IN.
  - `IRIN`: Instruction Register IN.
  - `RAMIN`: RAM IN. En aquest cas, no utilitza cap registre intern, sinó que guarda les dades a la posició adreçada pel registre d'adreça (MAR).
  - `RAIN`: A Register IN.
  - `RBIN`: B Register IN.
  - `OUT`: Mòdul de sortida IN. El nom pot semblar confús, però es tracta de llegir les dades del bus i guardar-les al registre del mòdul de sortida. La lògica de conversió de binari a decimal llegeix les dades d'aquest registre constantment.
  - `PCIN`: Program Counter IN. Aquesta senyal és especial, ja que és la que permet fer salts dins el programa. Com ja he explicat, el maquinari permet fer salts incondicionals (`JMP`) i salts condicionals (quan el resultat de l'última operació és negatiu o més gran que 255). Les senyals de control `JMP`, `JMC` i `JMN` es combinen amb certs bits al mòdul de suma i es genera la senyal `PCIN`. Per a més informació sobre quina és aquesta lògica, consulteu l'apèndix B.5.
- Senyals de lectura. La seva funció és permetre el mòdul mostrar les dades que emmagatzema al bus principal, perquè algun altre mòdul se les pugui copiar (escriure). Són les següents:
  - `RAMOUT`. RAM OUT.
  - `PCOUT`. Program Counter OUT.

  - `RAOUT`. A Register OUT.
  - `RBOUT`. B Register OUT.
  - `ALUOUT`. ALU OUT. El mòdul sumador no té cap registre per emmagatzemar el resultat, i per això, és força volàtil. Per aquest motiu, és freqüent que després de realitzar una operació, aquesta es guardi al registre A. Tot i que el sumador no és una ALU, he mantingut el nom per futurs dissenys que puguin contenir una ALU de veritat.

- Senyals de canvi de comportament. La seva funció és modificar el comportament per defecte del mòdul. Són les següents:

  - `HLT`. Halt. Atura el rellotge principal, de manera que el processador queda bloquejat. És la manera adequada d'acabar un programa. Un cop entrat en aquest estat, s'ha de reiniciar el processador per sortir-ne. Aquest és el comportament esperat.
  - `SUB`. Subtract. Permet al sumador realitzar restes (RA − RB) enlloc de sumes (RA + RB).
  - `PCINC`. Program Counter Increment. Incrementa en 1 el comptador de programa.
  - `Ca2`. Complement a dos. Permet al mòdul de sortida interpretar els valors en complement a 2, fent possible la representació de nombres negatius.

A.6 Microcodi

Cadascuna d'aquestes taules mostra les microinstruccions de cada instrucció (columna *Pas*) i quines senyals de control s'activen (columna *Senyals de control*).

A.6.1 Instrucció ADD

Pas	Senyals de control		
1	PCOUT	MAIN	
2	RAMOUT	IRIN	PCINC
3	PCOUT	MAIN	
4	RAMOUT	MAIN	PCINC
5	RAMOUT	RBIN	
6	ALUOUT	RAIN	RST

Taula A.3: Microinstruccions per la instrucció ADD.

A.6.2 Instrucció SUB

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	PCOUT	MAIN		
4	RAMOUT	MAIN	PCINC	
5	RAMOUT	RBIN		
6	ALUOUT	RAIN	SUB	RST

Taula A.4: Microinstruccions per la instrucció SUB.

A.6.3 Instrucció ADDI

Pas	Senyals de control		
1	PCOUT	MAIN	
2	RAMOUT	IRIN	PCINC
3	PCOUT	MAIN	
4	RAMOUT	RBIN	PCINC
5	ALUOUT	RAIN	RST

Taula A.5: Microinstruccions per la instrucció ADDI.

A.6.4 Instrucció SUBI

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	PCOUT	MAIN		
4	RAMOUT	RBIN	PCINC	
5	ALUOUT	RAIN	SUB	RST

Taula A.6: Microinstruccions per la instrucció SUBI.

A.6.5 Instrucció INC

Pas	Senyals de control		
1	PCOUT	MAIN	
2	RAMOUT	IRIN	PCINC
3	RAMOUT	RAIN	
4	PCOUT	MAIN	
5	RAMOUT	MAIN	
6	RAMOUT	RBIN	PCINC
7	ALUOUT	RAMIN	RST

Taula A.7: Microinstruccions per la instrucció INC. Per realitzar l'increment en 1 d'una posició de memòria, com que el registre IR no pot treure les dades al bus, m'aprofito que el codi d'instrucció per aquesta és 0x01. Per això al pas 3 utilitzo el codi d'instrucció per aconseguir el número 1.

A.6.6 Instrucció LDA

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	PCOUT	MAIN		
4	RAMOUT	MAIN		
5	RAMOUT	RAIN	PCINC	RST

Taula A.8: Microinstruccions per la instrucció LDA.

A.6.7 Instrucció LDAI

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	PCOUT	MAIN		
4	RAMOUT	RAIN	PCINC	RST

Taula A.9: Microinstruccions per la instrucció LDAI.

A.6.8 Instrucció STA

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	PCOUT	MAIN		
4	RAMOUT	MAIN		
5	RAMIN	RAOUT	PCINC	RST

Taula A.10: Microinstruccions per la instrucció STA.

A.6.9 Instrucció LDB

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	PCOUT	MAIN		
4	RAMOUT	MAIN		
5	RAMOUT	RBIN	PCINC	RST

Taula A.11: Microinstruccions per la instrucció LDB.

A.6.10 Instrucció LDBI

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	PCOUT	MAIN		
4	RAMOUT	RBIN	PCINC	RST

Taula A.12: Microinstruccions per la instrucció LDBI.

A.6.11 Instrucció STB

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	PCOUT	MAIN		
4	RAMOUT	MAIN		
5	RAMIN	RBOUT	PCINC	RST

Taula A.13: Microinstruccions per la instrucció STB.

A.6.12 Instrucció MOVAB

Pas	Senyals de control		
1	PCOUT	MAIN	
2	RAMOUT	IRIN	PCINC
3	RAOUT	RBIN	RST

Taula A.14: Microinstruccions per la instrucció MOVAB.

A.6.13 Instrucció MOVBA

Pas	Senyals de control		
1	PCOUT	MAIN	
2	RAMOUT	IRIN	PCINC
3	RBOUT	RAIN	RST

Taula A.15: Microinstruccions per la instrucció MOVBA.

A.6.14 Instrucció OUTS

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	RAOUT	OUT	Ca2	RST

Taula A.16: Microinstruccions per la instrucció OUTS.

A.6.15 Instrucció OUTU

Pas	Senyals de control		
1	PCOUT	MAIN	
2	RAMOUT	IRIN	PCINC
3	RAOUT	OUT	RST

Taula A.17: Microinstruccions per la instrucció OUTU.

A.6.16 Instrucció JMP

Pas	Senyals de control		
1	PCOUT	MAIN	
2	RAMOUT	IRIN	PCINC
3	PCOUT	MAIN	
4	RAMOUT	JMP	RST

Taula A.18: Microinstruccions per la instrucció JMP. La lògica de salt decidirà saltar, generant la senyal de control PCIN per actualitzar el comptador de programa.

A.6.17 Instrucció JMC

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	PCOUT	MAIN		
4	RAMOUT	JMC	PCINC	RST

Taula A.19: Microinstruccions per la instrucció JMC. La lògica de salt decidirà si saltar depenent del resultat de la última operació i generarà la senyal de control PCIN per actualitzar el comptador de programa.

A.6.18 Instrucció JMN

Pas	Senyals de control			
1	PCOUT	MAIN		
2	RAMOUT	IRIN	PCINC	
3	PCOUT	MAIN		
4	RAMOUT	JMN	PCINC	RST

Taula A.20: Microinstruccions per la instrucció JMN. La lògica de salt decidirà si saltar depenent del resultat de la última operació i generarà la senyal de control PCIN per actualitzar el comptador de programa.

A.6.19 Instrucció HLT

Pas	Senyals de control		
1	PCOUT	MAIN	
2	RAMOUT	IRIN	PCINC
3	HLT	RST	

Taula A.21: Microinstruccions per la instrucció HLT.

A.6.20 Instrucció NOP

Pas	Senyals de control		
1	PCOUT	MAIN	
2	RAMOUT	IRIN	PCINC

Taula A.22: Microinstruccions per la instrucció NOP.

Apèndix B

Esquemàtics del processador de 8 bits

Els esquemàtics estan creats amb el programa Eagle i es troben adjunts a la memòria.

B.1 Rellotge principal

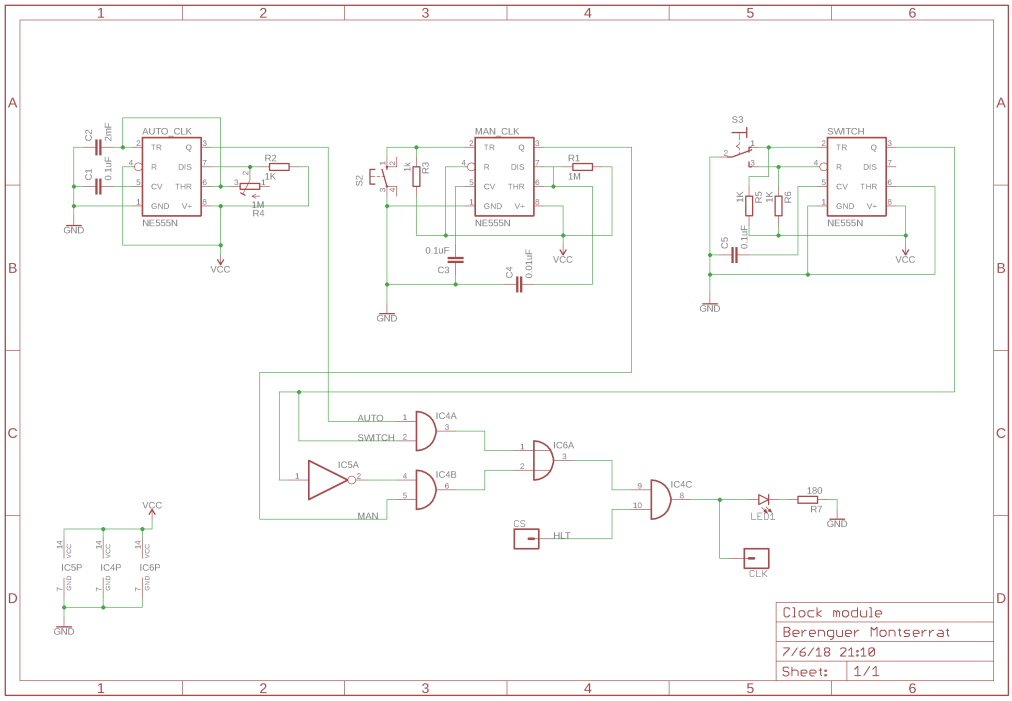

Figura B.1: Esquemàtic del rellotge principal.

B.2 Registres

B.2.1 Registre A (RA)

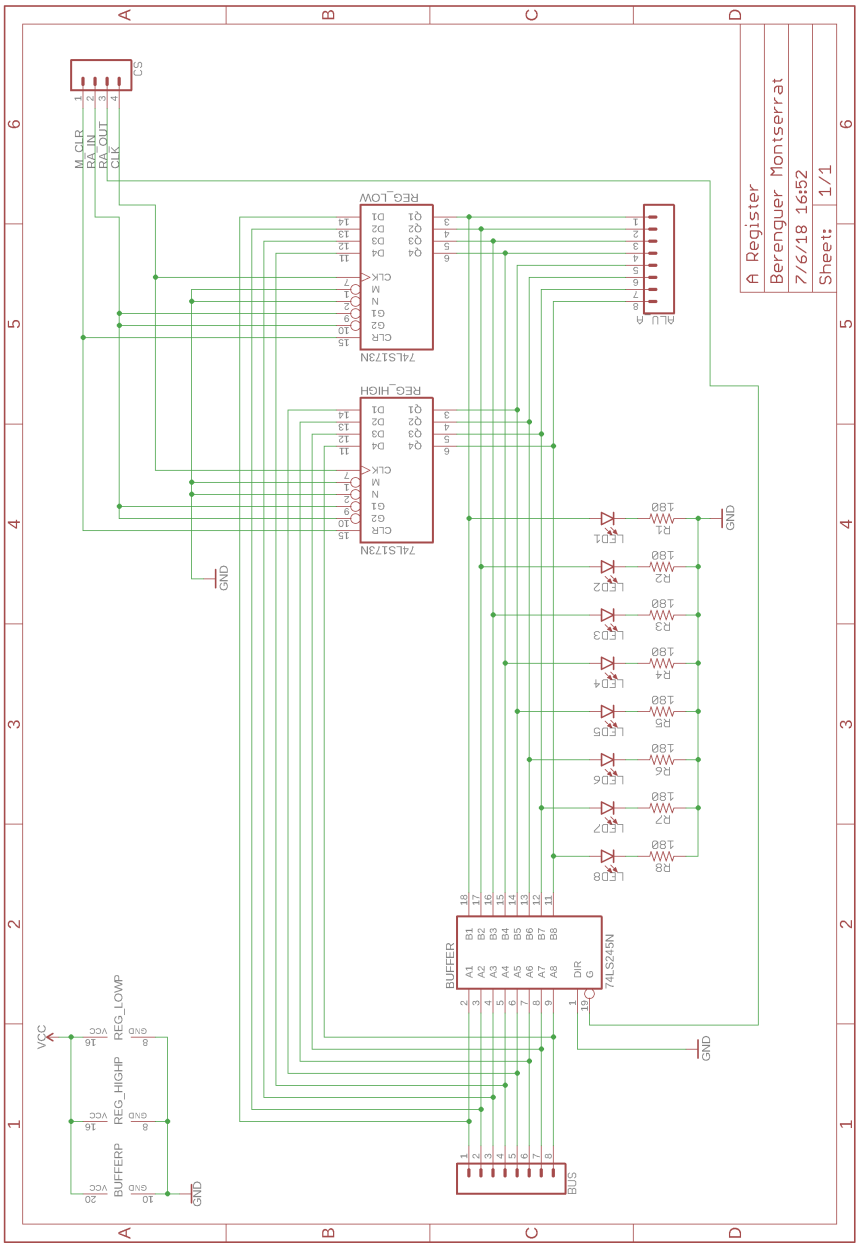

Figura B.2: Esquemàtic del registre A.

B.2.2 Registre B (RB)

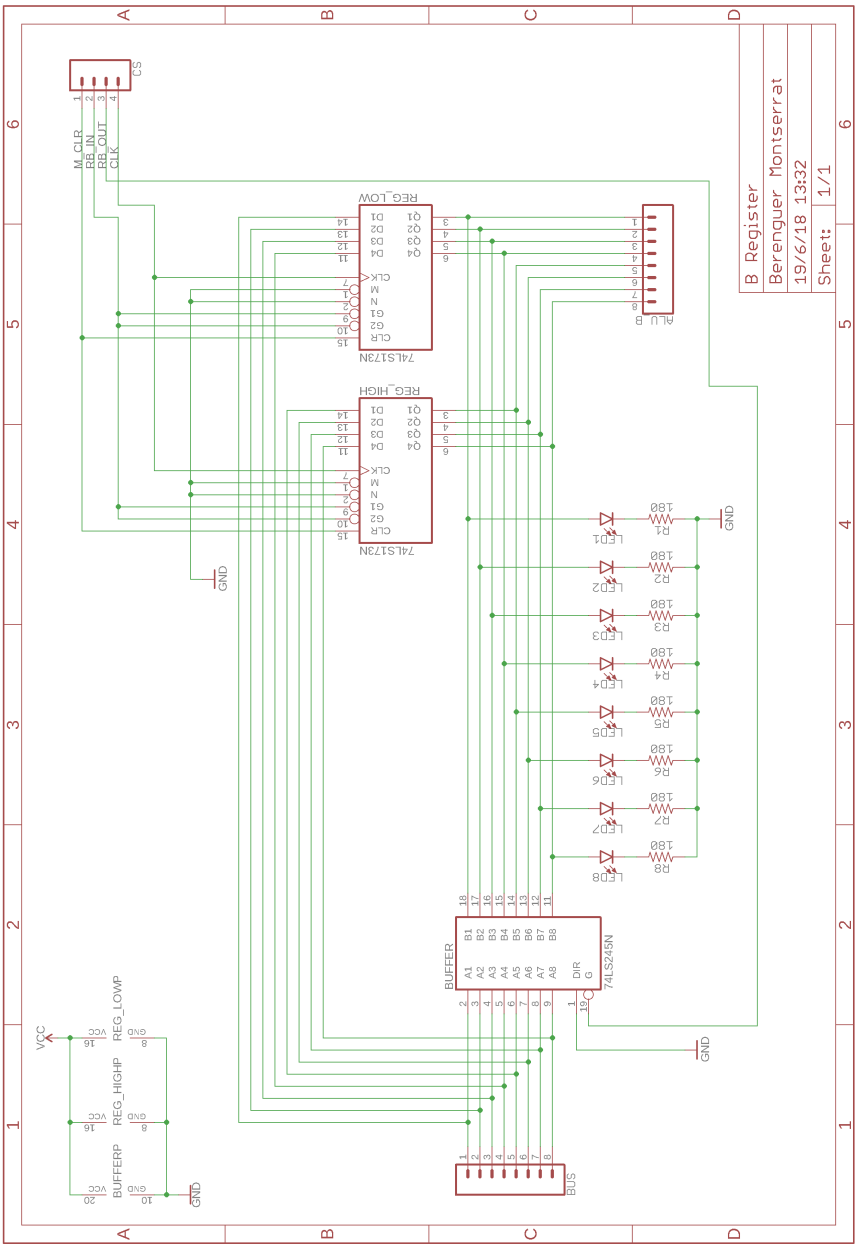

Figura B.3: Esquemàtic del registre B.

B.2.3 Registre d'instruccions (IR)

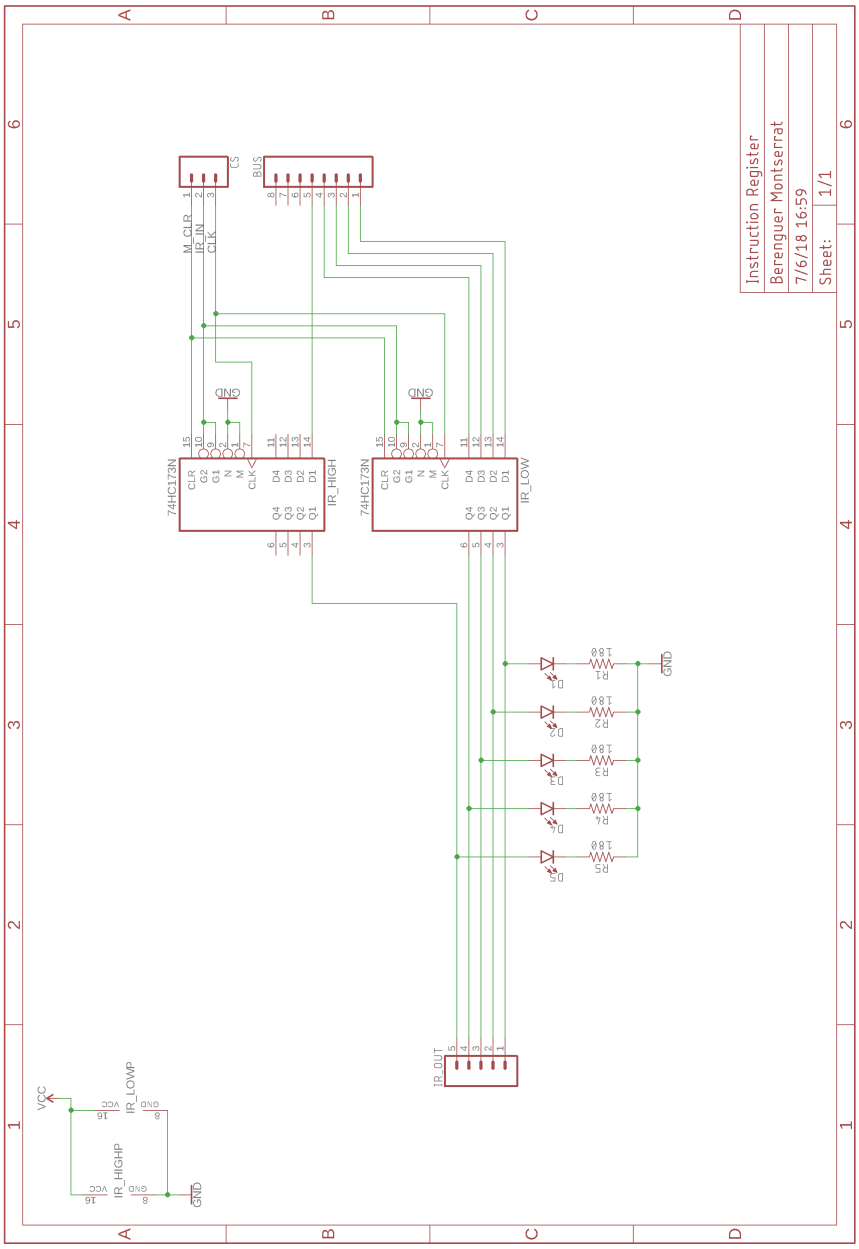

Figura B.4: Esquemàtic del registre d'instruccions.

B.2.4 Registre de sortida

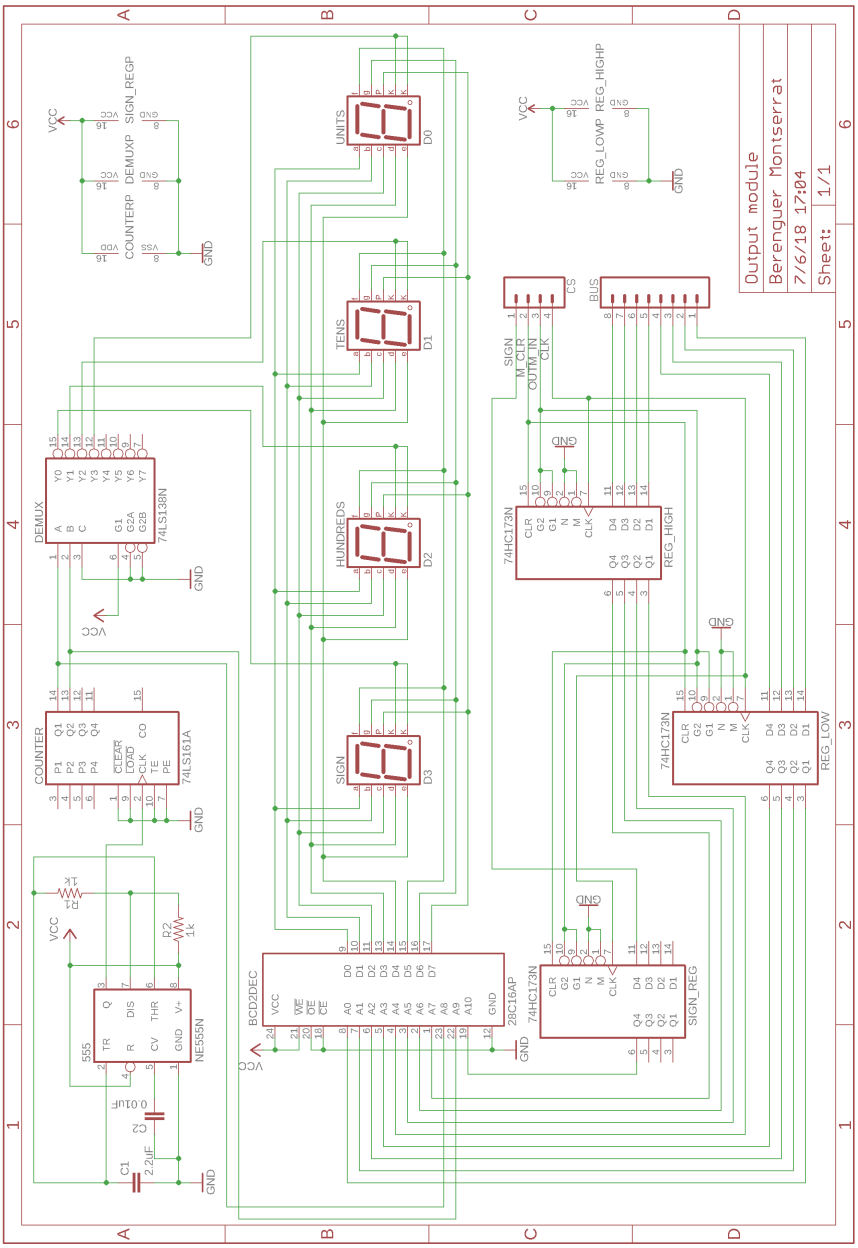

Figura B.5: Esquemàtic del registre de sortida i la lògica extra per mostrar nombres en binari naturalo Ca2 en decimal en les pantalles de 7 segments.

B.3 Comptador de programa (PC)

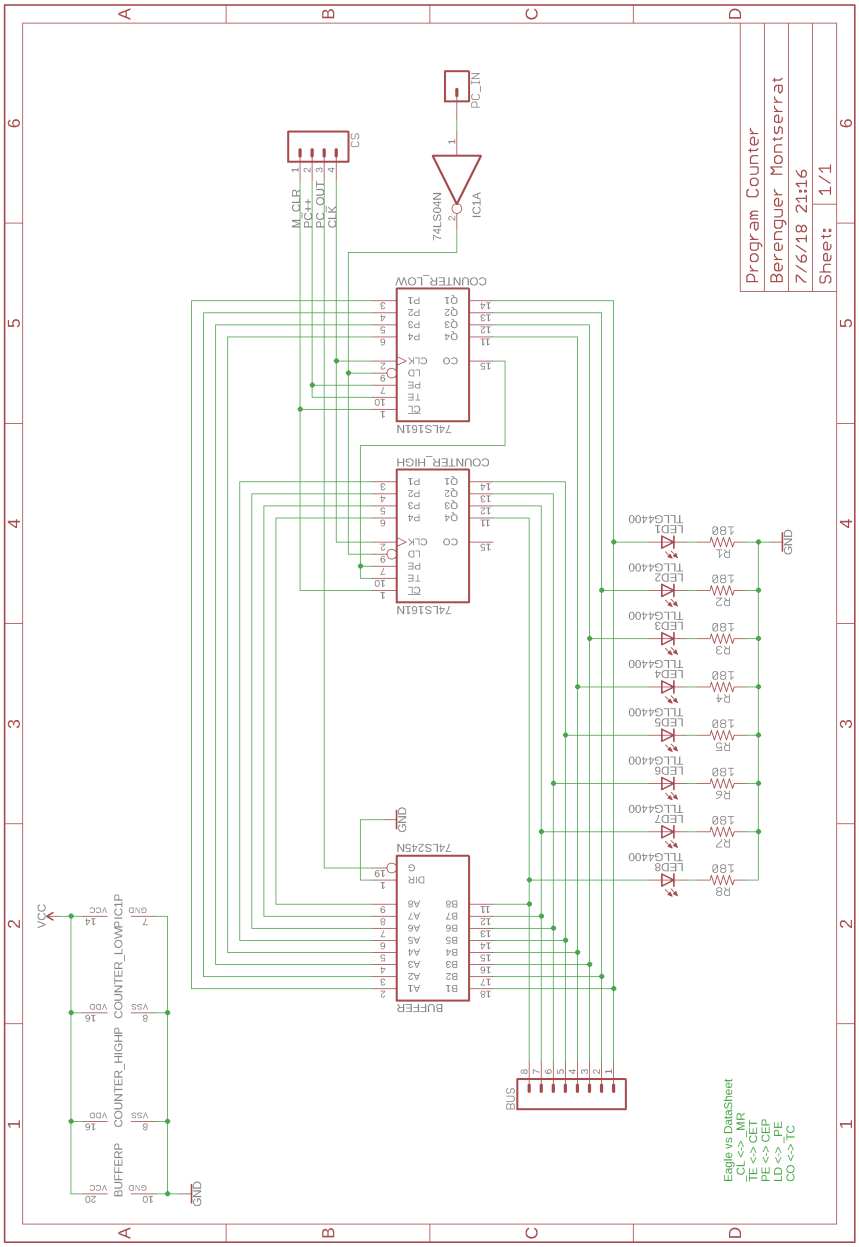

Figura B.6: Esquemàtic del comptador de programa.

B.4 Mòdul de memòria

B.4.1 Registre d'adreça de memòria (MAR)

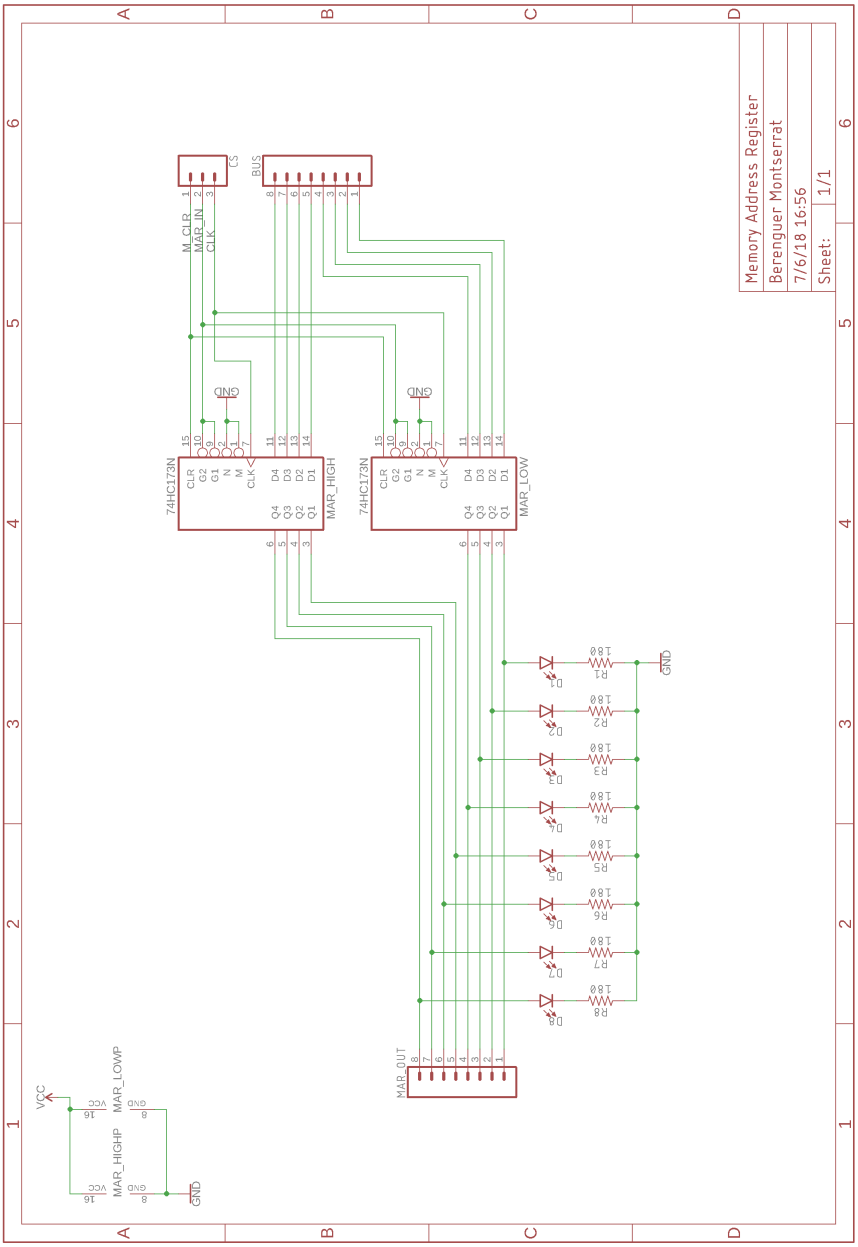

Figura B.7: Esquemàtic del registre d'adreça de memòria.

B.4.2 Memòria RAM

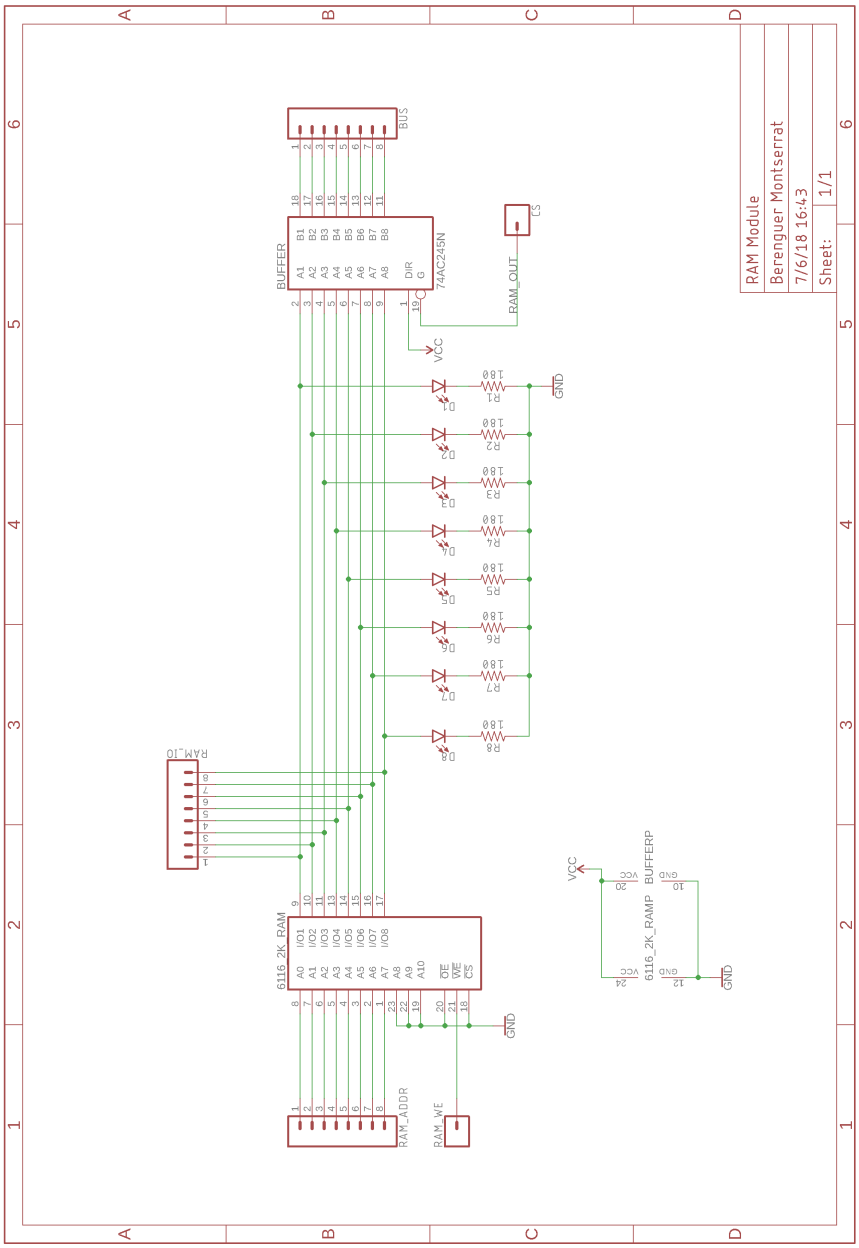

Figura B.8: Esquemàtic de la memòria RAM.

B.4.3 Programador

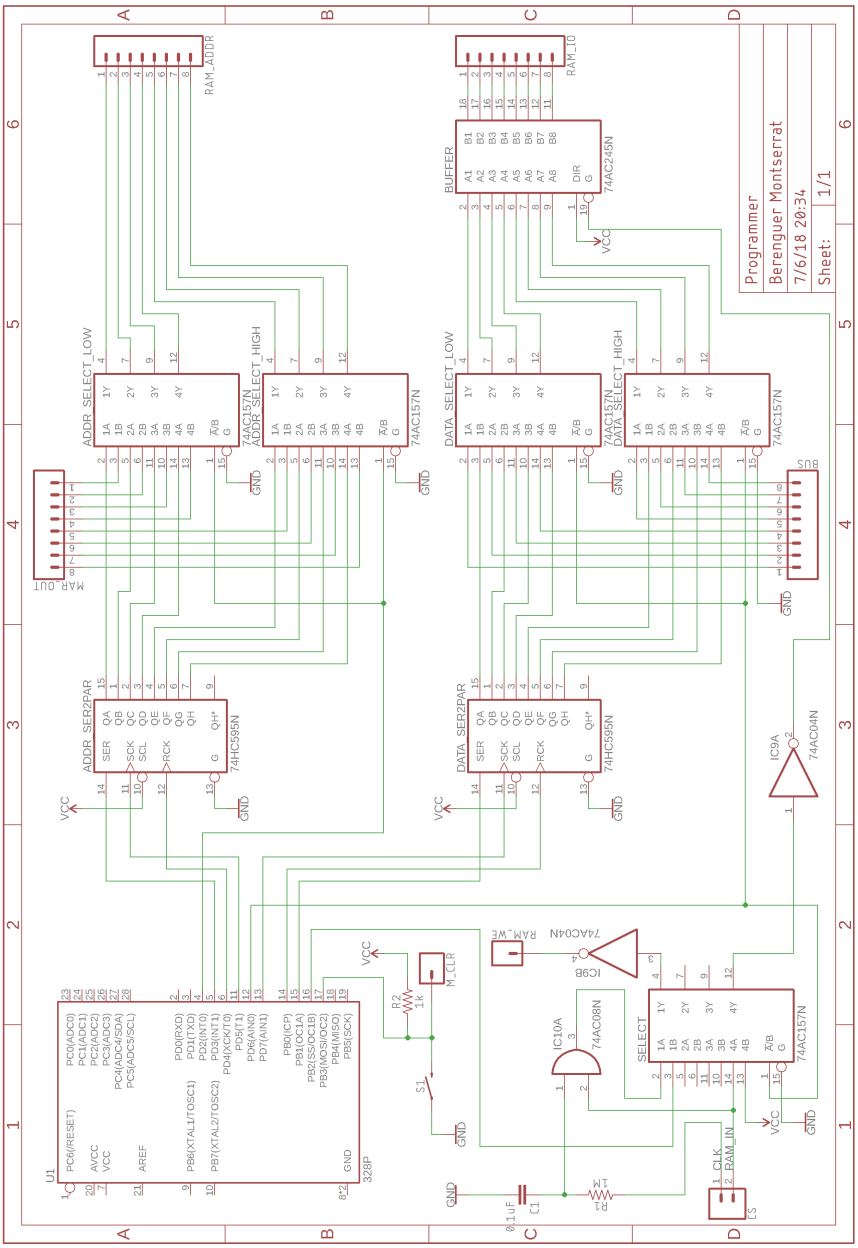

Figura B.9: Esquemàtic del programador del processador.

B.5 Sumador

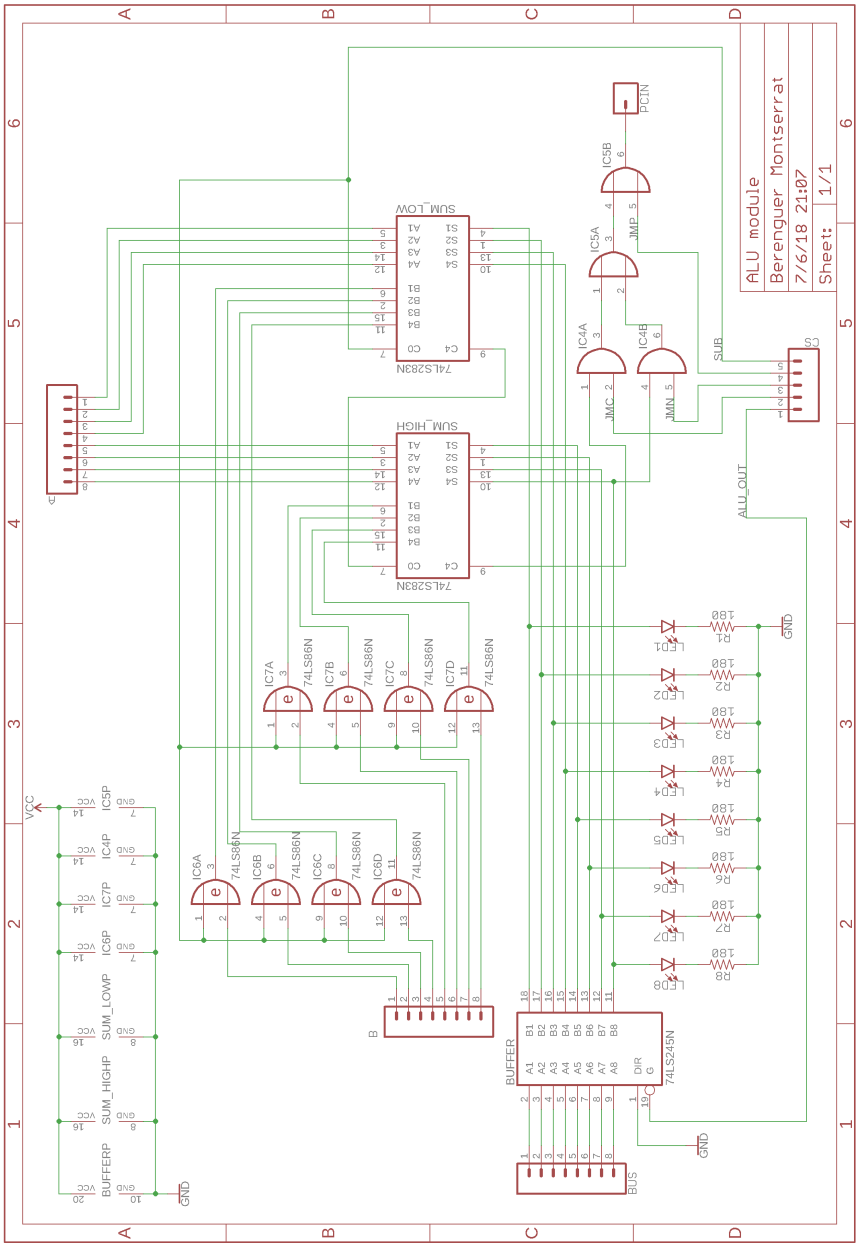

Figura B.10: Esquemàtic del sumador.

B.6 Lògica de control

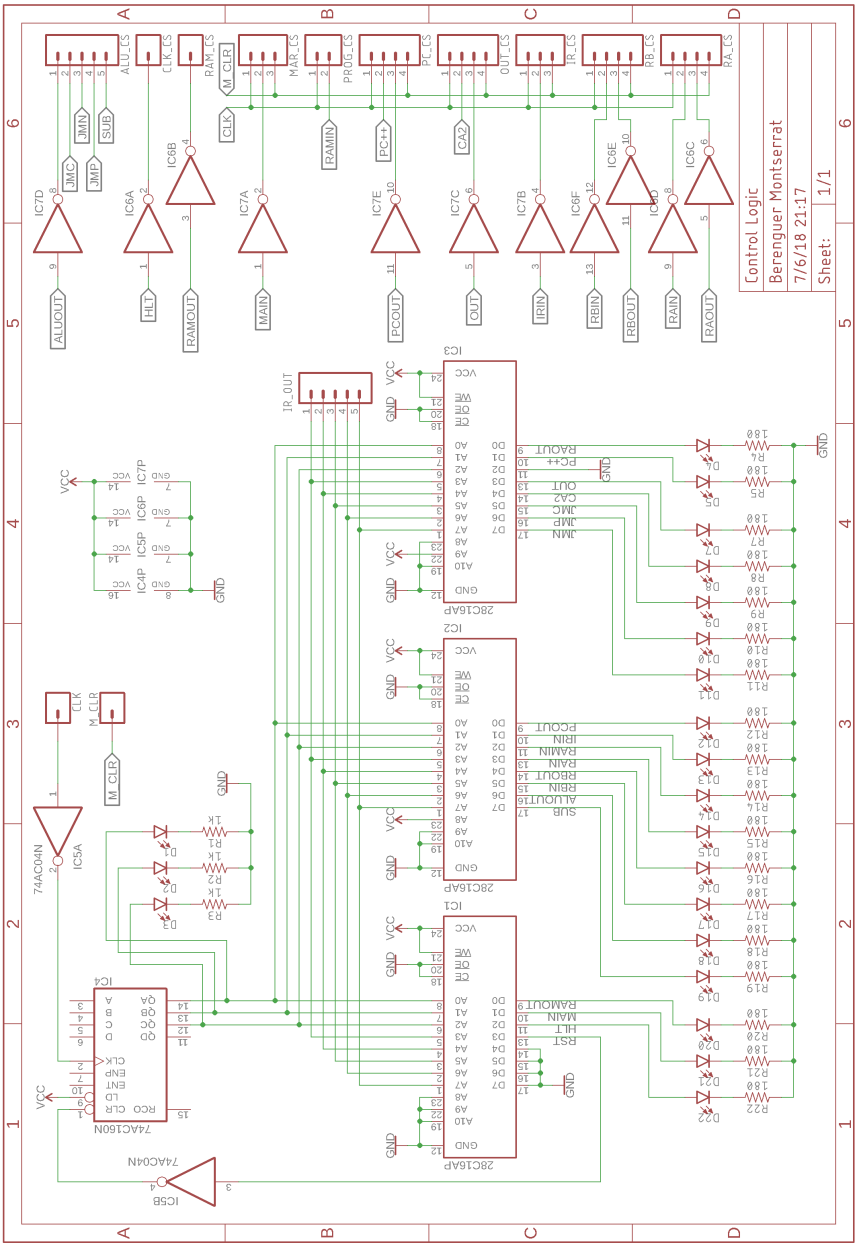

Figura B.11: Esquemàtic de la lògica de control.

Apèndix C

Context sobre les arquitectures utilitzades

C.1 Programació multi-fil a la CPU

La programació concurrent s'acostuma a realitzar en múltiples línies d'execució paral·lela. Hi ha dues maneres d'aconseguir-ho: mitjançant processos i mitjançant fils. La programació concurrent utilitzant processos s'acostuma a utilitzar quan aquests processos resideixen en diferents màquines, que és el cas contrari de la programació concurrent utilitzant fils.

Un fil d'execució és una línia d'execució paral·lela dins d'un mateix procés. Tot procés comença essent executat emprant un sol fil. Aquest en pot crear més i decidir quin codi han d'executar. La gestió dels fils la porta el sistema operatiu, per tant, no es pot assumir cap ordre ni prioritat entre els fils d'un mateix procés ni entre fils de diferents processos.

El mitjà de comunicació dels fils és l'espai de memòria del mateix procés, que és compartit. Cada fil, però, té la seva pila i els propis registres del processador.

El mecanisme que fa servir el sistema operatiu per gestionar els fils, a grans trets, és assignar-li a cadascun una llesca de temps perquè pugui ser executat. Un cop se li ha acabat el temps, el sistema operatiu n'escull un altre i li assigna una altra llesca de temps. Aquest fet permetia als primers processadors, que només tenien un nucli d'execució, executar més d'un programa a la vegada i donar a l'usuari la sensació de simultaneïtat. Però amb l'arribada de nous processadors amb múltiples unitats d'execució, aquest pseudo-paral·lelisme va convertir-se en paral·lelisme real. I és aquest fet el que aprofitaré per optimitzar la part de la creació de la llista d'índexs.

C.2 Instruccions vectorials a la CPU

Les instruccions SIMD (*Single Instruction Multiple Data*) o instruccions vectorials permeten aconseguir un paral·lelisme a nivell de dades aplicant la mateixa operació a múltiples elements al mateix instant. Aquest tipus de paral·lelisme difereix de la concurrència a nivell de fil, ja que no comporta cap gestió extra per part del sistema operatiu.

Aquest tipus de paral·lelisme s'aconsegueix gràcies a un maquinari específic que els processadors implementen. A grans trets, es tracta d'un banc de registres de 64, 128 o 256 bits (registres XMM) i un conjunt d'ALUs sincronitzades que operen amb les múltiples dades emmagatzemades dins de cada registre.

La primera implementació en processadors per ordinadors personals va ser l'any 1996 amb el conjunt d'instruccions d'Intel anomenat MMX. Els registres tenien una mida de 64 bits. Més tard, Intel va introduir un nou conjunt d'instruccions, SSE (*Streaming SIMD Extension*), que afegeix la possibilitat de treballar amb nombres de coma flotant sense dependre de la unitat principal de la CPU (FPU). Posteriorment, s'han anat fent millores i ampliacions amb els conjunts d'instruccions SSE2, SSE3, SSE4.1 i SSE4.2. Finalment, cap a l'any 2011, s'introdueixen els conjunts d'instruccions AVX i AVX2, que incrementen la mida dels registres de 128 bits (SSE) a 256 i 512 bits, augmentant la quantitat de dades processables al mateix temps i el conjunt d'operacions possibles.

Per esbrinar si aquests conjunts d'instruccions eren realment eficients i senzills d'utilitzar, vaig realitzar un petit *benchmark*, on havia de calcular 2^{29} arrels quadrades (en precisió simple). Vaig implementar-ho en sèrie, usant concurrència (OpenMP) i fent servir instruccions SIMD. Els resultats parlen per sí sols: el càlcul és 16 vegades més ràpid utilitzant instruccions SIMD que utilitzant concurrència en 4 fils.

L'avantatge principal de les instruccions SIMD és que proporcionen paral·lelisme a nivell de dades, no a nivell de fil o procés, eliminant tota la gestió extra que el sistema operatiu ha de realitzar. També, en màquines on el nombre de nuclis és limitat, ja sigui per construcció o per rendiment, l'ús d'instruccions SIMD proporciona un increment en rendiment utilitzant només un sol fil d'execució.

Per altra banda, aquestes instruccions presenten certes limitacions. La limitació que més diferencia les instruccions SIMD d'OpenMP i l'ús de targetes gràfiques (següent apartat), és l'absència de suport natiu per a la divergència; en altres paraules, tots els elements del vector es processaran de la mateixa manera.

C.3 Entorn OpenCL

OpenCL (*Open Computing Language*) és un estàndard obert, multiplataforma i lliure de drets de programació paral·lela pels diferents processadors que es poden trobar en ordinadors

personals, servidors, dispositius mòbils i sistemes encastats. És una API (*Application Programming Interface*) utilitzada per programar sistemes heterogenis (diferents processadors i targetes gràfiques en un mateix sistema). Utilitza el llenguatge C (c99). Permet aconseguir un gran paral·lelisme a nivell de dades. El grup *Khronos* n'és el principal mantenidor.

Aquest estàndard especifica una sèrie de directives que els dissenyadors de maquinari hauran de complir perquè tal maquinari pugui ser compatible amb OpenCL. Actualment, la majoria de processadors d'ordinadors personals i targetes gràfiques ja siguin integrades o discretes, l'implementen, fent que OpenCL sigui una eina molt utilitzada a l'hora de realitzar programes paral·lels per a diferents arquitectures.

Els fabricants més coneguts de processadors en el sector d'ordinadors personals, Intel i AMD, proporcionen un suport complet a nivell de maquinari i de programari. Pel que fa a targetes gràfiques, Radeon proporciona suport complet, mentre que NVIDIA només proveeix la capacitat d'executar codi OpenCL a la majoria dels seus productes, ja que la resta d'eines només són compatibles amb CUDA, un estàndard propietari semblant a OpenCL.

OpenCL organitza el maquinari de la següent manera: el grup més gran és anomenat plataforma (*platform*) i acostuma a ser el fabricant del maquinari, per exemple, Intel o Nvidia. Dins de cada plataforma, hi ha un seguit de dispositius (*devices*), que és el maquinari en sí, com per exemple, un processador o una targeta gràfica. Un cop s'ha seleccionat el dispositiu, s'ha de crear un context, que serà l'eina de comunicació entre l'hoste i el dispositiu. Després es crea una cua de comandes (*command_queue*), on l'hoste anirà encuant peticions al dispositiu, ja sigui per copiar dades o iniciar un càlcul.

Per defecte, el codi que el dispositiu executa (*kernel*) es compila en temps d'execució, d'aquesta manera es pot seleccionar el compilador adequat segons el dispositiu que s'hagi triat. Tot i així, es pot guanyar temps d'inicialització pre-compilant els *kernels*.

En l'entorn OpenCL, tots els dispositius es tracten de manera asíncrona amb l'hoste, ja que generalment, l'hoste i el dispositiu són maquinaris diferents que resideixen a la mateixa màquina. Aquest fet fa que s'hagi de tenir en compte col·locar punts de sincronització hoste-dispositiu, però permet que tant l'hoste com el dispositiu treballin de manera paral·lela. És en aquest punt on el concepte *collaborative computing* (computació col·laborativa) entra en joc.

Apèndix D

NVIDIA Jetson TX2

D.1 Diagrama de blocs

Al bloc d'NVIDIA Developers, hi ha l'entrada *NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge* [9], on es presenta el producte amb les seves especificacions i capacitats, amb un munt d'enllaços i referències pels desenvolupadors per començar a treballar amb aquesta plataforma. Una de les imatges que he fet més cops d'ull és la de la figura D.1, on es veu el diagrama de blocs de la Jetson TX2.

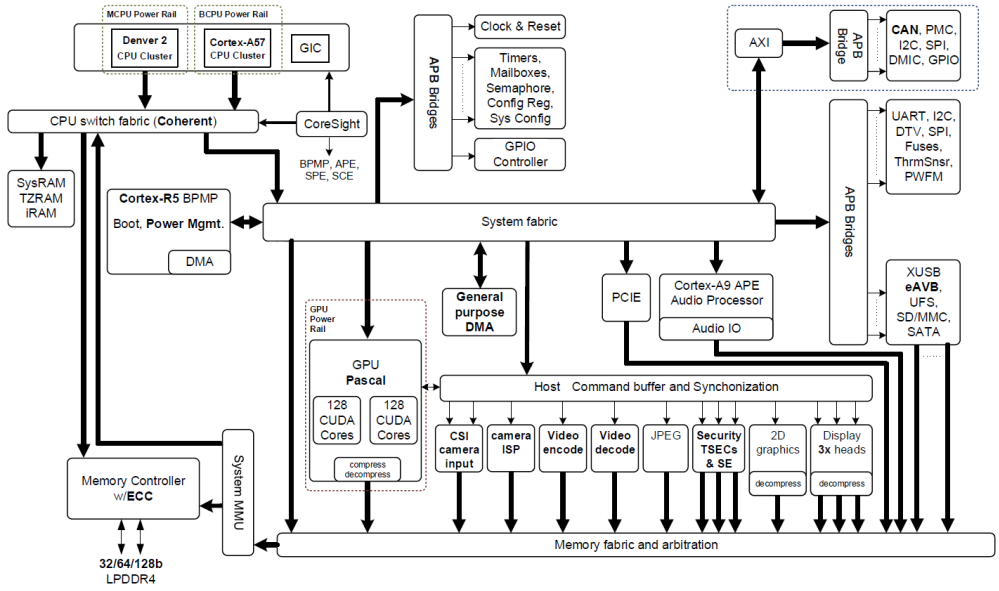

Figura D.1: Diagrama de blocs de la Jetson TX2. Es poden observar els diferents components de maquinari que formen el xip i els busos que els connecten.